Python Programming Exam Summary

Comprehensive Study Guide for Pen-and-Paper Examination

Author: Manus AI
Date: December 2024
Course: Python Programming and Data Structures

Table of Contents

Python Basics

Python is a high-level, interpreted programming language that emphasizes code readability and simplicity. Understanding the fundamental concepts of Python is crucial for success in any programming examination. This section covers the essential building blocks that form the foundation of Python programming.

Definition of Script Language

Python is classified as a script language, which means it is interpreted rather than compiled. Unlike compiled languages such as C++ or Java, Python code is executed line by line by the Python interpreter at runtime. This characteristic provides several advantages including rapid development cycles, interactive programming capabilities, and platform independence. The interpreter reads Python source code and converts it into bytecode, which is then executed by the Python Virtual Machine (PVM).

The interpreted nature of Python means that you can test code snippets immediately in an interactive environment, making it ideal for prototyping and educational purposes. However, this also means that syntax errors are only discovered when the problematic line is executed, unlike compiled languages where all syntax errors are caught during compilation.

Data Types and Operations

Python supports several built-in data types that are fundamental to programming. Understanding these types and their operations is essential for exam success.

Numeric Types

Python provides three numeric types: integers (int), floating-point numbers (float), and complex numbers (complex). Integers in Python 3 have unlimited precision, meaning they can be arbitrarily large. Floating-point numbers follow the IEEE 754 standard and have limited precision. Complex numbers consist of a real and imaginary part.

```
# Integer operations
x = 42
y = -17
result = x + y  # Addition: 25
result = x * y  # Multiplication: -714
result = x // y  # Floor division: -3
result = x % y  # Modulo: -1
result = x ** 2  # Exponentiation: 1764

# Float operations
a = 3.14
b = 2.71
result = a / b  # Division: 1.158671586715867
result = round(a, 1)  # Rounding: 3.1

# Complex numbers
c = 3 + 4j
```

```
d = 1 - 2j
result = c + d  # (4+2j)
```

String Type

Strings in Python are immutable sequences of Unicode characters. They can be created using single quotes, double quotes, or triple quotes for multi-line strings. String operations include concatenation, repetition, slicing, and various built-in methods.

```
 # String creation
name = "Python"
message = 'Hello World'
multiline = """This is a
multi-line string"""

# String operations
full_message = name + " " + message  # Concatenation
repeated = "Ha" * 3  # Repetition: "HaHaHa"
substring = name[0:3]  # Slicing: "Pyt"
length = len(name)  # Length: 6
```

Boolean Type

Boolean values in Python are True and False (note the capitalization). Boolean operations include logical AND, OR, and NOT operations. Python uses short-circuit evaluation for boolean expressions.

```
 # Boolean operations
is_valid = True
is_complete = False
result = is_valid and is_complete  # False
result = is_valid or is_complete   # True
result = not is_valid              # False

# Comparison operations return boolean values
x = 10
y = 20
result = x < y    # True
result = x == y   # False
result = x != y   # True
```

Collection Types

Python provides several built-in collection types: lists, tuples, dictionaries, and sets. Each has distinct characteristics and use cases.

Lists are ordered, mutable collections that can contain elements of different types:

```
numbers = [1, 2, 3, 4, 5]
mixed = [1, "hello", 3.14, True]
numbers.append(6)         # Add element
numbers.insert(0, 0)      # Insert at position
numbers.remove(3)         # Remove first occurrence
element = numbers.pop()   # Remove and return last element
```

Tuples are ordered, immutable collections:

```
coordinates = (10, 20)
rgb = (255, 128, 0)
# Tuples cannot be modified after creation
# coordinates[0] = 15  # This would raise an error
```

Dictionaries are unordered collections of key-value pairs:

```
student = {
    "name": "Alice",
    "age": 20,
    "grade": "A"
}
student["email"] = "alice@example.com"  # Add new key-value pair
name = student["name"]                   # Access value
age = student.get("age", 0)             # Safe access with
default
```

Sets are unordered collections of unique elements:

```
numbers = {1, 2, 3, 4, 5}
numbers.add(6)            # Add element
numbers.remove(3)         # Remove element
numbers.discard(10)       # Remove if exists (no error if not
found)
```

Variable Assignment and Naming

Variable assignment in Python is straightforward and follows specific naming conventions. Variables are created when first assigned and do not need explicit type declarations.

```
# Valid variable names
age = 25
```

```
first_name = "John"
_private_var = "hidden"
counter2 = 0

# Invalid variable names (would cause syntax errors)
# 2counter = 0      # Cannot start with digit
# first-name = ""   # Cannot contain hyphens
# class = "value"   # Cannot use reserved keywords
```

Python follows the PEP 8 style guide for naming conventions: - Variables and functions use snake_case - Constants use UPPER_CASE - Classes use PascalCase - Private attributes start with underscore

Operators and Expressions

Python provides various operators for different operations:

Arithmetic Operators: - `+` (addition), `-` (subtraction), `*` (multiplication) - `/` (division), `//` (floor division), `%` (modulo) - `**` (exponentiation)

Comparison Operators: - `==` (equal), `!=` (not equal) - `<`, `>`, `<=`, `>=` (relational comparisons) - `is` (identity), `in` (membership)

Logical Operators: - `and`, `or`, `not`

Assignment Operators: - `=` (assignment) - `+=`, `-=`, `*=`, `/=` (compound assignment)

Understanding operator precedence is crucial for writing correct expressions. Python follows standard mathematical precedence rules, with parentheses having the highest precedence, followed by exponentiation, multiplication and division, and finally addition and subtraction.

Control Structures I - Conditional Statements

Conditional statements allow programs to make decisions and execute different code paths based on specific conditions. Python provides several forms of conditional statements that are essential for controlling program flow.

Basic if Statement

The fundamental conditional statement in Python is the `if` statement. It evaluates a boolean expression and executes the indented code block only if the condition is True.

```python
age = 18
if age >= 18:
    print("You are eligible to vote")
    print("Welcome to the voting booth")
```

The syntax requires a colon after the condition and proper indentation for the code block. Python uses indentation (typically 4 spaces) to define code blocks, unlike languages that use curly braces.

if-else Statement

The `if-else` statement provides an alternative execution path when the condition is False:

```python
temperature = 25
if temperature > 30:
    print("It's hot outside")
    clothing = "shorts and t-shirt"
else:
    print("It's not too hot")
    clothing = "regular clothes"
```

if-elif-else Statement

For multiple conditions, Python provides the `elif` (else if) clause, allowing you to test multiple conditions in sequence:

```python
score = 85
if score >= 90:
    grade = "A"
    print("Excellent work!")
elif score >= 80:
    grade = "B"
    print("Good job!")
elif score >= 70:
    grade = "C"
    print("Satisfactory")
elif score >= 60:
    grade = "D"
    print("Needs improvement")
else:
    grade = "F"
    print("Failed")
```

The conditions are evaluated in order, and only the first True condition's block is executed. Once a condition is met, the remaining elif and else blocks are skipped.

Nested Conditional Statements

Conditional statements can be nested within other conditional statements to handle complex decision-making scenarios:

```
weather = "sunny"
temperature = 22
if weather == "sunny":
    if temperature > 25:
        activity = "go to the beach"
    elif temperature > 15:
        activity = "go for a walk"
    else:
        activity = "stay inside and read"
else:
    if temperature > 20:
        activity = "indoor sports"
    else:
        activity = "watch movies"
```

Shorthand if Statement

Python supports a shorthand version of the if statement for simple conditions:

```
# Traditional if statement
if age >= 18:
    status = "adult"

# Shorthand version
status = "adult" if age >= 18 else "minor"
```

This ternary operator is useful for simple assignments but should be used sparingly to maintain code readability.

Switch-Case Alternative

Python does not have a traditional switch-case statement like some other languages. However, similar functionality can be achieved using if-elif chains or dictionary mappings:

```
# Using if-elif chain
day_number = 3
```

```python
if day_number == 1:
    day_name = "Monday"
elif day_number == 2:
    day_name = "Tuesday"
elif day_number == 3:
    day_name = "Wednesday"
elif day_number == 4:
    day_name = "Thursday"
elif day_number == 5:
    day_name = "Friday"
elif day_number == 6:
    day_name = "Saturday"
elif day_number == 7:
    day_name = "Sunday"
else:
    day_name = "Invalid day"

# Using dictionary mapping (more efficient for many cases)
day_mapping = {
    1: "Monday",
    2: "Tuesday",
    3: "Wednesday",
    4: "Thursday",
    5: "Friday",
    6: "Saturday",
    7: "Sunday"
}
day_name = day_mapping.get(day_number, "Invalid day")
```

Boolean Expressions and Logical Operators

Conditional statements rely on boolean expressions that evaluate to True or False. Understanding how to construct complex boolean expressions is crucial:

```python
 age = 25
income = 50000
has_job = True
credit_score = 750

# Complex boolean expression
if (age >= 18 and age <= 65) and (income > 30000 or has_job) and
credit_score > 600:
    loan_approved = True
    print("Loan application approved")
else:
    loan_approved = False
    print("Loan application denied")
```

Truthiness and Falsiness

Python has specific rules about what values are considered True or False in boolean contexts:

Falsy values: - `False` - `None` - `0` (zero) - `0.0` (zero float) - `""` (empty string) - `[]` (empty list) - `{}` (empty dictionary) - `()` (empty tuple) - `set()` (empty set)

Truthy values: - Everything else, including non-zero numbers, non-empty strings, and non-empty collections

```python
# Examples of truthiness
name = ""
if name:  # False, because empty string is falsy
    print("Name provided")
else:
    print("No name provided")

numbers = [1, 2, 3]
if numbers:  # True, because non-empty list is truthy
    print("List has elements")
```

Common Patterns and Best Practices

When writing conditional statements, several patterns and best practices should be followed:

1. **Use clear and descriptive conditions:** Make boolean expressions readable and self-documenting.

2. **Avoid deep nesting:** Excessive nesting makes code hard to read. Consider using early returns or guard clauses.

3. **Use parentheses for clarity:** When combining multiple conditions, use parentheses to make the order of operations explicit.

4. **Consider using `in` for multiple equality checks:**

```python
# Instead of multiple or conditions
if day == "Saturday" or day == "Sunday":
    is_weekend = True

# Use membership testing
if day in ["Saturday", "Sunday"]:
    is_weekend = True
```

Understanding conditional statements is fundamental to programming logic and forms the basis for more complex control structures and algorithms.

Control Structures II - Loops

Loops are fundamental control structures that allow programs to execute code repeatedly. Python provides two main types of loops: `while` loops and `for` loops. Understanding when and how to use each type is essential for efficient programming and exam success.

While Loops

The `while` loop executes a block of code repeatedly as long as a specified condition remains True. It is particularly useful when the number of iterations is not known in advance.

Basic While Loop Syntax

```python
counter = 0
while counter < 5:
    print(f"Counter value: {counter}")
    counter += 1  # Important: update the condition variable
```

The while loop consists of three essential components: 1. **Initialization:** Setting up the loop variable before the loop begins 2. **Condition:** The boolean expression that determines whether the loop continues 3. **Update:** Modifying the loop variable to eventually make the condition False

Infinite Loops and Loop Control

Without proper updates to the condition variable, while loops can become infinite loops:

```python
# Infinite loop (avoid this!)
# while True:
#     print("This will run forever")

# Controlled infinite loop with break
while True:
    user_input = input("Enter 'quit' to exit: ")
    if user_input.lower() == 'quit':
        break
    print(f"You entered: {user_input}")
```

While Loop with Else Clause

Python's while loops can include an optional `else` clause that executes when the loop completes normally (not via a `break` statement):

```python
number = 10
while number > 0:
    print(number)
    number -= 1
else:
    print("Countdown complete!")  # Executes after loop finishes
```

For Loops

The `for` loop is used to iterate over sequences (strings, lists, tuples, dictionaries, sets) or other iterable objects. It is generally preferred when the number of iterations is known or when working with collections.

Basic For Loop Syntax

```python
# Iterating over a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(f"I like {fruit}")

# Iterating over a string
word = "Python"
for letter in word:
    print(letter)

# Iterating over a range
for i in range(5):  # 0, 1, 2, 3, 4
    print(f"Number: {i}")
```

The range() Function

The `range()` function is commonly used with for loops to generate sequences of numbers:

```python
# range(stop) - from 0 to stop-1
for i in range(5):
    print(i)  # 0, 1, 2, 3, 4

# range(start, stop) - from start to stop-1
for i in range(2, 8):
    print(i)  # 2, 3, 4, 5, 6, 7

# range(start, stop, step) - with custom step
for i in range(0, 10, 2):
    print(i)  # 0, 2, 4, 6, 8

# Reverse range
for i in range(10, 0, -1):
    print(i)  # 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

Iterating Over Dictionaries

When iterating over dictionaries, you can access keys, values, or both:

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

# Iterate over keys (default behavior)
for name in student_grades:
    print(f"Student: {name}")

# Explicitly iterate over keys
for name in student_grades.keys():
    print(f"Student: {name}")

# Iterate over values
for grade in student_grades.values():
    print(f"Grade: {grade}")

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

Enumerate Function

The `enumerate()` function provides both the index and value when iterating over sequences:

```python
colors = ["red", "green", "blue"]
for index, color in enumerate(colors):
    print(f"Index {index}: {color}")

# Starting enumerate from a different number
for index, color in enumerate(colors, start=1):
    print(f"Color {index}: {color}")
```

Zip Function

The `zip()` function allows parallel iteration over multiple sequences:

```python
names = ["Alice", "Bob", "Charlie"]
ages = [25, 30, 35]
cities = ["New York", "London", "Tokyo"]

for name, age, city in zip(names, ages, cities):
    print(f"{name} is {age} years old and lives in {city}")
```

Loop Control Statements

Python provides several statements to control loop execution:

Break Statement

The `break` statement immediately exits the loop:

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

Continue Statement

The `continue` statement skips the rest of the current iteration and moves to the next:

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for num in numbers:
    if num % 2 == 0:  # Skip even numbers
        continue
    print(num)  # Prints 1, 3, 5, 7, 9
```

Pass Statement

The `pass` statement is a null operation used as a placeholder:

```python
for i in range(10):
    if i < 5:
        pass  # Placeholder - do nothing
    else:
        print(i)  # Prints 5, 6, 7, 8, 9
```

Nested Loops

Loops can be nested within other loops to handle multi-dimensional data or complex iterations:

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

# Working with 2D lists
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
```

For Loop with Else Clause

Similar to while loops, for loops can have an else clause that executes when the loop completes normally:

```
numbers = [2, 4, 6, 8, 10]
for num in numbers:
    if num % 2 != 0:  # Check for odd number
        print("Found an odd number")
        break
else:
    print("All numbers are even")  # Executes if no break
occurred
```

Common Loop Patterns

Several patterns frequently appear in programming and exams:

Accumulator Pattern

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

```python
# Building a string
words = ["Hello", "world", "from", "Python"]
sentence = ""
for word in words:
    sentence += word + " "
print(sentence.strip())
```

Counter Pattern

```python
# Counting specific elements
text = "Hello World"
vowel_count = 0
for char in text.lower():
    if char in "aeiou":
        vowel_count += 1
print(f"Vowels: {vowel_count}")
```

Search Pattern

```python
# Finding an element
numbers = [10, 20, 30, 40, 50]
target = 30
found = False
for num in numbers:
    if num == target:
        found = True
        break
if found:
    print(f"Found {target}")
else:
    print(f"{target} not found")
```

Maximum/Minimum Pattern

```python
# Finding maximum value
numbers = [45, 23, 67, 12, 89, 34]
max_value = numbers[0]  # Initialize with first element
for num in numbers[1:]:  # Start from second element
    if num > max_value:
        max_value = num
print(f"Maximum: {max_value}")
```

Performance Considerations

Understanding the performance characteristics of different loop approaches is important:

1. **For loops are generally faster** than while loops for iterating over sequences
2. **List comprehensions** (covered later) are often faster than equivalent for loops
3. **Built-in functions** like `sum()`, `max()`, `min()` are optimized and preferred over manual loops when applicable

```python
# Slower approach
numbers = [1, 2, 3, 4, 5]
total = 0
for num in numbers:
    total += num

# Faster approach
total = sum(numbers)
```

Understanding loops is crucial for solving algorithmic problems and is frequently tested in programming exams through various scenarios including data processing, pattern generation, and iterative calculations.

IDE and Remote Development

Integrated Development Environments (IDEs) and remote development tools are essential for efficient Python programming. Understanding these tools and their capabilities is important for both practical programming and exam scenarios where you might need to debug code or understand development workflows.

Python IDEs and Editors

Popular Python IDEs

PyCharm is a comprehensive IDE developed by JetBrains that offers extensive features for Python development including intelligent code completion, debugging, testing, and version control integration. It comes in both Professional and Community editions.

Visual Studio Code (VS Code) is a lightweight, extensible editor that becomes a powerful Python IDE with the Python extension. It offers features like IntelliSense, debugging, Git integration, and extensive customization options.

IDLE is Python's built-in IDE that comes with the standard Python installation. While basic, it provides a simple environment for learning and testing Python code with features like syntax highlighting and an interactive shell.

Jupyter Notebook is an interactive environment particularly popular for data science and research. It allows combining code, text, and visualizations in a single document.

Key IDE Features

Modern Python IDEs provide several essential features:

Syntax Highlighting makes code more readable by coloring different elements (keywords, strings, comments) differently.

Code Completion suggests possible completions as you type, reducing errors and speeding up development.

Debugging Tools allow you to set breakpoints, step through code, and inspect variables during execution.

Version Control Integration provides seamless integration with Git and other version control systems.

Project Management helps organize files and dependencies within larger projects.

Interactive Python Environment

Python REPL

The Python Read-Eval-Print Loop (REPL) is an interactive environment where you can execute Python statements immediately:

```
 >>> x = 10
>>> y = 20
>>> print(x + y)
30
>>> def greet(name):
...     return f"Hello, {name}!"
...
>>> greet("World")
'Hello, World!'
```

The REPL is excellent for testing small code snippets, exploring libraries, and learning Python interactively.

IPython

IPython is an enhanced interactive Python shell that provides additional features:

```
In [1]: import math

In [2]: math.pi
Out[2]: 3.141592653589793

In [3]: ?math.sqrt  # Get help on function
```

IPython offers features like magic commands, better error messages, and integration with Jupyter notebooks.

Remote Development

SSH and Remote Servers

Remote development allows you to write and execute code on remote servers while using local development tools. This is particularly useful for:

- Accessing more powerful computing resources
- Working with production-like environments
- Collaborating with team members
- Accessing specialized software or data

```
# Connecting to remote server via SSH
ssh username@remote-server.com

# Running Python scripts remotely
python3 my_script.py

# Transferring files
scp local_file.py username@remote-server.com:/path/to/
destination/
```

VS Code Remote Development

VS Code's Remote Development extensions allow you to:

- Connect to remote servers via SSH
- Work with code in containers
- Develop in Windows Subsystem for Linux (WSL)

The experience feels like local development while actually running on remote resources.

Virtual Environments

Virtual environments are crucial for managing Python dependencies and avoiding conflicts between projects.

Creating Virtual Environments

```
# Using venv (Python 3.3+)
python -m venv myproject_env

# Activating the environment
# On Windows:
myproject_env\Scripts\activate
# On macOS/Linux:
source myproject_env/bin/activate

# Installing packages
pip install requests numpy pandas

# Deactivating
deactivate
```

Requirements Files

Requirements files help manage project dependencies:

```
# Generate requirements file
pip freeze > requirements.txt

# Install from requirements file
pip install -r requirements.txt
```

Example requirements.txt:

```
requests==2.28.1
numpy==1.21.0
pandas==1.3.3
```

Package Management

pip - Python Package Installer

pip is the standard package manager for Python:

```
# Install a package
pip install package_name

# Install specific version
pip install package_name==1.2.3

# Upgrade a package
pip install --upgrade package_name

# Uninstall a package
pip uninstall package_name

# List installed packages
pip list

# Show package information
pip show package_name
```

conda

conda is an alternative package manager that can handle both Python and non-Python dependencies:

```
# Create environment
conda create --name myenv python=3.9

# Activate environment
conda activate myenv

# Install packages
conda install numpy pandas matplotlib

# List environments
conda env list
```

Debugging Techniques

Print Debugging

The simplest debugging technique involves adding print statements:

```python
def calculate_average(numbers):
    print(f"Input numbers: {numbers}")  # Debug print
    total = sum(numbers)
    print(f"Total: {total}")  # Debug print
    count = len(numbers)
    print(f"Count: {count}")  # Debug print
    average = total / count
    print(f"Average: {average}")  # Debug print
    return average
```

Using the Debugger

Python's built-in `pdb` debugger allows interactive debugging:

```
import pdb

def problematic_function(x, y):
    pdb.set_trace()  # Breakpoint
    result = x / y
    return result

# When executed, this will drop into the debugger
```

Common pdb commands: - `n` (next line) - `s` (step into function) - `c` (continue execution) - `l` (list current code) - `p variable_name` (print variable value) - `q` (quit debugger)

IDE Debugging

Modern IDEs provide graphical debugging interfaces:

1. Set breakpoints by clicking in the margin
2. Start debugging mode
3. Use step over, step into, step out buttons
4. Inspect variables in the variables panel
5. Evaluate expressions in the console

Code Quality Tools

Linting

Linters analyze code for potential errors and style issues:

```
# Install pylint
pip install pylint

# Run pylint on a file
pylint my_script.py
```

Code Formatting

Automatic code formatters ensure consistent style:

```
# Install black formatter
pip install black

# Format a file
black my_script.py

# Format entire project
black .
```

Type Checking

Type checkers like mypy can catch type-related errors:

```
# Install mypy
pip install mypy

# Check types
mypy my_script.py
```

Version Control Integration

Git Basics

Understanding Git is essential for collaborative development:

```
# Initialize repository
git init

# Add files to staging
git add file.py
git add .  # Add all files

# Commit changes
git commit -m "Add new feature"

# Check status
git status

# View history
git log

# Create branch
git branch feature-branch
git checkout feature-branch

# Merge branch
git checkout main
git merge feature-branch
```

.gitignore for Python

A typical .gitignore file for Python projects:

```
 # Byte-compiled / optimized / DLL files
__pycache__/
*.py[cod]
*$py.class

# Virtual environments
venv/
env/
ENV/

# IDE files
.vscode/
.idea/
*.swp
*.swo

# OS files
.DS_Store
Thumbs.db
```

Best Practices for Development Environment

1. **Use virtual environments** for each project to avoid dependency conflicts
2. **Keep dependencies updated** but test thoroughly after updates
3. **Use version control** for all projects, even small ones
4. **Write clear commit messages** that explain what and why
5. **Use consistent code formatting** across the project
6. **Set up linting** to catch errors early
7. **Learn keyboard shortcuts** for your IDE to improve productivity
8. **Backup your work** regularly and use cloud-based repositories

Understanding development environments and tools is crucial for efficient programming and is often tested in practical programming scenarios where you need to set up, debug, or maintain Python projects.

File Operations

File operations are fundamental to many programming tasks, allowing programs to read data from files, write results to files, and persist information between program executions. Python provides comprehensive file handling capabilities that are frequently tested in programming exams.

Opening and Closing Files

Basic File Operations

Python uses the `open()` function to create file objects that can be used to read from or write to files:

```python
# Opening a file for reading
file = open("data.txt", "r")
content = file.read()
file.close()  # Always close files when done

# Opening a file for writing
file = open("output.txt", "w")
file.write("Hello, World!")
file.close()
```

File Modes

The second parameter to `open()` specifies the mode:

- `"r"` - Read mode (default)
- `"w"` - Write mode (overwrites existing content)
- `"a"` - Append mode (adds to end of file)
- `"x"` - Exclusive creation (fails if file exists)
- `"b"` - Binary mode (e.g., "rb", "wb")
- `"t"` - Text mode (default)
- `"+"` - Read and write mode

```python
# Different file modes
read_file = open("input.txt", "r")
write_file = open("output.txt", "w")
append_file = open("log.txt", "a")
binary_file = open("image.jpg", "rb")
read_write_file = open("data.txt", "r+")
```

Context Managers and the with Statement

The `with` statement provides a clean way to handle files by automatically closing them when done:

```
# Using with statement (recommended approach)
with open("data.txt", "r") as file:
    content = file.read()
    print(content)
# File is automatically closed here

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

The `with` statement ensures that files are properly closed even if an exception occurs, making it the preferred method for file operations.

Reading Files

Reading Entire File

```
# Read entire file as a string
with open("story.txt", "r") as file:
    content = file.read()
    print(content)

# Read entire file as a list of lines
with open("story.txt", "r") as file:
    lines = file.readlines()
    for line in lines:
        print(line.strip())  # strip() removes newline
characters
```

Reading Line by Line

```
# Method 1: Using readline()
with open("data.txt", "r") as file:
    line = file.readline()
    while line:
        print(line.strip())
        line = file.readline()

# Method 2: Iterating over file object (most Pythonic)
```

```python
with open("data.txt", "r") as file:
    for line in file:
        print(line.strip())

# Method 3: Using readlines()
with open("data.txt", "r") as file:
    for line in file.readlines():
        print(line.strip())
```

Reading Specific Amounts

```python
 # Read specific number of characters
with open("data.txt", "r") as file:
    chunk = file.read(10)  # Read first 10 characters
    print(chunk)

# Read in chunks (useful for large files)
with open("large_file.txt", "r") as file:
    while True:
        chunk = file.read(1024)  # Read 1KB at a time
        if not chunk:
            break
        process_chunk(chunk)
```

Writing Files

Writing Strings

```python
 # Write a single string
with open("output.txt", "w") as file:
    file.write("Hello, World!\n")
    file.write("This is a second line.\n")

# Write multiple lines
lines = ["First line\n", "Second line\n", "Third line\n"]
with open("output.txt", "w") as file:
    file.writelines(lines)
```

Appending to Files

```python
 # Append to existing file
with open("log.txt", "a") as file:
    file.write("New log entry\n")

# Append with timestamp
import datetime
```

```
with open("log.txt", "a") as file:
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:
%M:%S")
    file.write(f"[{timestamp}] Application started\n")
```

File Paths and Directory Operations

Working with Paths

```
import os

# Get current working directory
current_dir = os.getcwd()
print(f"Current directory: {current_dir}")

# Join paths (platform-independent)
file_path = os.path.join("data", "files", "input.txt")
print(f"File path: {file_path}")

# Check if file exists
if os.path.exists("data.txt"):
    print("File exists")
else:
    print("File not found")

# Check if path is file or directory
if os.path.isfile("data.txt"):
    print("It's a file")
elif os.path.isdir("data"):
    print("It's a directory")
```

Using pathlib (Modern Approach)

```
from pathlib import Path

# Create path objects
file_path = Path("data") / "files" / "input.txt"
print(f"File path: {file_path}")

# Check existence
if file_path.exists():
    print("File exists")

# Get file information
if file_path.is_file():
    print(f"File size: {file_path.stat().st_size} bytes")
    print(f"File name: {file_path.name}")
    print(f"File extension: {file_path.suffix}")
    print(f"Parent directory: {file_path.parent}")

# Create directories
output_dir = Path("output")
output_dir.mkdir(exist_ok=True)  # Create if doesn't exist
```

Error Handling with Files

Common File Exceptions

```
try:
    with open("nonexistent.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("File not found")
except PermissionError:
    print("Permission denied")
except IOError:
    print("I/O error occurred")
except Exception as e:
    print(f"Unexpected error: {e}")
```

Robust File Operations

```
def safe_read_file(filename):
    """Safely read a file with error handling."""
    try:
        with open(filename, "r", encoding="utf-8") as file:
            return file.read()
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found")
        return None
    except PermissionError:
        print(f"Error: Permission denied for '{filename}'")
        return None
    except UnicodeDecodeError:
        print(f"Error: Cannot decode '{filename}' as UTF-8")
        return None

def safe_write_file(filename, content):
    """Safely write to a file with error handling."""
    try:
        with open(filename, "w", encoding="utf-8") as file:
            file.write(content)
        return True
    except PermissionError:
        print(f"Error: Permission denied for '{filename}'")
        return False
```

```python
    except IOError as e:
        print(f"Error writing to '{filename}': {e}")
        return False
```

Working with CSV Files

Reading CSV Files

```python
import csv

# Reading CSV with csv module
with open("data.csv", "r") as file:
    csv_reader = csv.reader(file)
    header = next(csv_reader)  # Read header row
    print(f"Header: {header}")

    for row in csv_reader:
        print(row)

# Reading CSV as dictionaries
with open("data.csv", "r") as file:
    csv_reader = csv.DictReader(file)
    for row in csv_reader:
        print(f"Name: {row['name']}, Age: {row['age']}")
```

Writing CSV Files

```python
import csv

# Writing CSV data
data = [
    ["Name", "Age", "City"],
    ["Alice", 25, "New York"],
    ["Bob", 30, "London"],
    ["Charlie", 35, "Tokyo"]
]

with open("output.csv", "w", newline="") as file:
    csv_writer = csv.writer(file)
    csv_writer.writerows(data)

# Writing CSV from dictionaries
people = [
    {"name": "Alice", "age": 25, "city": "New York"},
    {"name": "Bob", "age": 30, "city": "London"}
]

with open("people.csv", "w", newline="") as file:
    fieldnames = ["name", "age", "city"]
    csv_writer = csv.DictWriter(file, fieldnames=fieldnames)
    csv_writer.writeheader()
    csv_writer.writerows(people)
```

Working with JSON Files

Reading JSON

```
import json

# Reading JSON file
with open("data.json", "r") as file:
    data = json.load(file)
    print(data)

# Reading JSON string
json_string = '{"name": "Alice", "age": 25, "city": "New York"}'
data = json.loads(json_string)
print(data["name"])
```

Writing JSON

```
import json

# Writing to JSON file
data = {
    "students": [
        {"name": "Alice", "grade": 85},
        {"name": "Bob", "grade": 92}
    ]
}

with open("students.json", "w") as file:
    json.dump(data, file, indent=2)

# Converting to JSON string
json_string = json.dumps(data, indent=2)
print(json_string)
```

File Processing Patterns

Processing Large Files

```
def process_large_file(filename):
    """Process a large file line by line to save memory."""
```

```python
    line_count = 0
    word_count = 0

    with open(filename, "r") as file:
        for line in file:
            line_count += 1
            words = line.split()
            word_count += len(words)

            # Process line here
            if line_count % 1000 == 0:
                print(f"Processed {line_count} lines")

    return line_count, word_count
```

File Backup and Versioning

```python
import shutil
from datetime import datetime

def backup_file(filename):
    """Create a backup of a file with timestamp."""
    if os.path.exists(filename):
        timestamp = 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
    else:
        print(f"File {filename} not found")
        return None
```

Configuration Files

```python
# Reading configuration from a simple text file
def read_config(filename):
    """Read configuration from key=value format file."""
    config = {}
    try:
        with open(filename, "r") as file:
            for line in file:
                line = line.strip()
                if line and not line.startswith("#"):  # Skip
empty lines and comments
                    key, value = line.split("=", 1)
                    config[key.strip()] = value.strip()
    except FileNotFoundError:
        print(f"Config file {filename} not found")
    return config
```

```
# Example config.txt:
# database_host=localhost
# database_port=5432
# debug_mode=true
```

Best Practices for File Operations

1. **Always use context managers** (`with` statement) for file operations
2. **Specify encoding explicitly** when working with text files
3. **Handle exceptions appropriately** for robust file operations
4. **Use pathlib** for modern path manipulation
5. **Close files properly** to avoid resource leaks
6. **Validate file paths** before operations
7. **Use appropriate file modes** for your specific needs
8. **Consider memory usage** when working with large files
9. **Backup important files** before modifying them
10. **Use standard formats** (CSV, JSON) for structured data

File operations are essential for data persistence and are commonly tested in programming exams through scenarios involving data processing, configuration management, and file-based data storage and retrieval.

Functions I - Introduction

Functions are fundamental building blocks in Python programming that allow code reuse, organization, and modularity. Understanding how to define, call, and work with functions is essential for programming success and frequently appears in exams.

Function Definition and Syntax

Basic Function Definition

A function in Python is defined using the `def` keyword, followed by the function name, parameters in parentheses, and a colon. The function body is indented:

```
def greet():
    """A simple function that prints a greeting."""
    print("Hello, World!")

# Calling the function
greet()  # Output: Hello, World!
```

Functions with Parameters

Functions can accept parameters (also called arguments) to make them more flexible:

```python
def greet_person(name):
    """Greet a specific person."""
    print(f"Hello, {name}!")

def add_numbers(a, b):
    """Add two numbers and print the result."""
    result = a + b
    print(f"{a} + {b} = {result}")

# Calling functions with arguments
greet_person("Alice")  # Output: Hello, Alice!
add_numbers(5, 3)      # Output: 5 + 3 = 8
```

Functions with Return Values

Functions can return values using the `return` statement:

```python
def calculate_area(length, width):
    """Calculate the area of a rectangle."""
    area = length * width
    return area

def get_full_name(first_name, last_name):
    """Combine first and last name."""
    full_name = f"{first_name} {last_name}"
    return full_name

# Using return values
rectangle_area = calculate_area(10, 5)
print(f"Area: {rectangle_area}")  # Output: Area: 50

name = get_full_name("John", "Doe")
print(name)  # Output: John Doe
```

Function Parameters and Arguments

Positional Arguments

Arguments are passed to functions in the order they are defined:

```python
def introduce(name, age, city):
    print(f"My name is {name}, I am {age} years old, and I live in {city}")
```

```
introduce("Alice", 25, "New York")
# Output: My name is Alice, I am 25 years old, and I live in New
York
```

Keyword Arguments

Arguments can be passed by specifying the parameter name:

```
def introduce(name, age, city):
    print(f"My name is {name}, I am {age} years old, and I live
in {city}")

# Using keyword arguments (order doesn't matter)
introduce(city="London", name="Bob", age=30)
introduce(name="Charlie", age=35, city="Tokyo")
```

Default Parameters

Functions can have default values for parameters:

```
def greet(name, greeting="Hello"):
    """Greet someone with a customizable greeting."""
    print(f"{greeting}, {name}!")

greet("Alice")              # Output: Hello, Alice!
greet("Bob", "Hi")          # Output: Hi, Bob!
greet("Charlie", greeting="Hey")  # Output: Hey, Charlie!
```

Mixing Parameter Types

When mixing different parameter types, the order must be: positional, default, args, *kwargs:

```
def complex_function(required_param, default_param="default",
*args, **kwargs):
    print(f"Required: {required_param}")
    print(f"Default: {default_param}")
    print(f"Args: {args}")
    print(f"Kwargs: {kwargs}")

complex_function("value1", "value2", "extra1", "extra2",
key1="val1", key2="val2")
```

Variable-Length Arguments

*args (Arbitrary Positional Arguments)

The `*args` parameter allows a function to accept any number of positional arguments:

```python
def sum_all(*numbers):
    """Calculate the sum of any number of arguments."""
    total = 0
    for num in numbers:
        total += num
    return total

result1 = sum_all(1, 2, 3)          # 6
result2 = sum_all(10, 20, 30, 40)   # 100
result3 = sum_all()                 # 0

def print_info(name, *hobbies):
    """Print name and hobbies."""
    print(f"Name: {name}")
    print("Hobbies:")
    for hobby in hobbies:
        print(f"  - {hobby}")

print_info("Alice", "reading", "swimming", "coding")
```

**kwargs (Arbitrary Keyword Arguments)

The `**kwargs` parameter allows a function to accept any number of keyword arguments:

```python
def create_profile(**info):
    """Create a user profile from keyword arguments."""
    print("User Profile:")
    for key, value in info.items():
        print(f"  {key}: {value}")

create_profile(name="Alice", age=25, city="New York",
occupation="Engineer")

def process_data(required_param, *args, **kwargs):
    """Function demonstrating all parameter types."""
    print(f"Required: {required_param}")
    print(f"Additional args: {args}")
    print(f"Keyword args: {kwargs}")

process_data("important", 1, 2, 3, debug=True, verbose=False)
```

Function Documentation

Docstrings

Docstrings provide documentation for functions and are accessible via the `help()` function:

```python
def calculate_bmi(weight, height):
    """
    Calculate Body Mass Index (BMI).

    Args:
        weight (float): Weight in kilograms
        height (float): Height in meters

    Returns:
        float: BMI value

    Example:
        >>> calculate_bmi(70, 1.75)
        22.857142857142858
    """
    bmi = weight / (height ** 2)
    return bmi

# Accessing docstring
print(calculate_bmi.__doc__)
help(calculate_bmi)
```

Local vs Global Variables

Local Variables

Variables defined inside a function are local to that function:

```python
def my_function():
    local_var = "I'm local"
    print(local_var)

my_function()  # Output: I'm local
# print(local_var)  # This would cause an error - local_var is
not accessible here
```

Global Variables

Variables defined outside functions are global and can be accessed from anywhere:

```python
global_var = "I'm global"

def access_global():
    print(global_var)  # Can read global variable

def modify_global():
    global global_var
    global_var = "Modified global"  # Need 'global' keyword to
modify

access_global()   # Output: I'm global
modify_global()
print(global_var) # Output: Modified global
```

The global Keyword

To modify a global variable inside a function, use the `global` keyword:

```python
counter = 0  # Global variable

def increment():
    global counter
    counter += 1

def get_counter():
    return counter

print(get_counter())  # 0
increment()
print(get_counter())  # 1
increment()
print(get_counter())  # 2
```

Function Return Values

Single Return Value

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

result = square(5)  # 25
```

Multiple Return Values

Functions can return multiple values as a tuple:

```python
def get_name_parts(full_name):
    """Split a full name into first and last name."""
    parts = full_name.split()
    first_name = parts[0]
    last_name = parts[-1]
    return first_name, last_name

first, last = get_name_parts("John Doe")
print(f"First: {first}, Last: {last}")

# Can also return as tuple
name_tuple = get_name_parts("Jane Smith")
print(name_tuple)  # ('Jane', 'Smith')
```

Early Returns

Functions can have multiple return statements:

```python
def check_grade(score):
    """Determine letter grade based on score."""
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 70:
        return "C"
    elif score >= 60:
        return "D"
    else:
        return "F"

grade = check_grade(85)  # "B"
```

Returning None

If a function doesn't explicitly return a value, it returns `None`:

```python
def print_message(message):
    print(message)
    # No return statement

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

Function Examples and Patterns

Mathematical Functions

```python
def factorial(n):
    """Calculate factorial of n."""
    if n <= 1:
        return 1
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result

def is_prime(n):
    """Check if a number is prime."""
    if n < 2:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True
```

String Processing Functions

```python
def count_vowels(text):
    """Count vowels in a string."""
    vowels = "aeiouAEIOU"
    count = 0
    for char in text:
        if char in vowels:
            count += 1
    return count

def reverse_words(sentence):
    """Reverse the order of words in a sentence."""
    words = sentence.split()
    reversed_words = words[::-1]
    return " ".join(reversed_words)
```

List Processing Functions

```python
def find_maximum(numbers):
    """Find the maximum value in a list."""
    if not numbers:
        return None
    max_val = numbers[0]
    for num in numbers[1:]:
        if num > max_val:
            max_val = num
    return max_val

def filter_even(numbers):
    """Return a list of even numbers."""
    even_numbers = []
    for num in numbers:
        if num % 2 == 0:
            even_numbers.append(num)
    return even_numbers
```

Function Best Practices

1. **Use descriptive names** that clearly indicate what the function does
2. **Keep functions small and focused** on a single task
3. **Use docstrings** to document function purpose, parameters, and return values
4. **Avoid global variables** when possible; prefer parameters and return values
5. **Use default parameters** for optional functionality
6. **Return meaningful values** rather than printing inside functions when possible
7. **Handle edge cases** appropriately (empty lists, None values, etc.)
8. **Use type hints** for better code documentation (covered in later sections)

Common Function Patterns in Exams

Input Validation

```python
def safe_divide(a, b):
    """Safely divide two numbers."""
    if b == 0:
        return None  # or raise an exception
    return a / b
```

Data Transformation

```python
def celsius_to_fahrenheit(celsius):
    """Convert Celsius to Fahrenheit."""
    fahrenheit = (celsius * 9/5) + 32
    return fahrenheit
```

List Operations

```python
def get_statistics(numbers):
    """Calculate basic statistics for a list of numbers."""
    if not numbers:
        return None

    total = sum(numbers)
    count = len(numbers)
    average = total / count
    minimum = min(numbers)
    maximum = max(numbers)

    return {
        'sum': total,
        'count': count,
        'average': average,
        'min': minimum,
        'max': maximum
    }
```

Understanding functions is crucial for writing modular, reusable code and is a fundamental concept that appears in various forms throughout programming exams, from basic function definition to complex parameter handling and return value manipulation.

String Operations and Formatting

String manipulation is a fundamental skill in Python programming, essential for text processing, data cleaning, and user interface development. Python provides extensive string operations and formatting capabilities that are frequently tested in programming exams.

String Basics and Properties

String Immutability

Strings in Python are immutable, meaning they cannot be changed after creation. Any operation that appears to modify a string actually creates a new string:

```python
 text = "Hello"
# text[0] = "h"  # This would raise an error
new_text = "h" + text[1:]  # Create new string: "hello"
```

String Creation

```python
 # Different ways to create strings
single_quotes = 'Hello World'
```

```
double_quotes = "Hello World"
triple_quotes = """This is a
multi-line string"""
raw_string = r"C:\Users\name\file.txt"  # Raw string
(backslashes not escaped)
```

String Methods

Case Conversion Methods

```
text = "Hello World"

# Case conversion
upper_text = text.upper()         # "HELLO WORLD"
lower_text = text.lower()         # "hello world"
title_text = text.title()         # "Hello World"
capitalize_text = text.capitalize()  # "Hello world"
swapcase_text = text.swapcase()   # "hELLO wORLD"

# Case checking
is_upper = text.isupper()         # False
is_lower = text.islower()         # False
is_title = text.istitle()         # True
```

String Searching and Checking

```
text = "Python Programming"

# Finding substrings
index = text.find("Pro")          # 7 (returns -1 if not found)
rindex = text.rfind("o")          # 15 (rightmost occurrence)
count = text.count("o")           # 2

# Checking string properties
starts = text.startswith("Py")    # True
ends = text.endswith("ing")       # True
is_alpha = text.isalpha()         # False (contains space)
is_digit = "123".isdigit()        # True
is_alnum = "abc123".isalnum()     # True
```

String Cleaning and Modification

```
text = "  Hello World  "

# Whitespace removal
stripped = text.strip()           # "Hello World"
```

```
left_stripped = text.lstrip()     # "Hello World  "
right_stripped = text.rstrip()    # "  Hello World"

# Character replacement
replaced = text.replace("World", "Python")  # "  Hello Python  "
translated = "hello".replace("l", "L")      # "heLLo"

# String splitting and joining
words = "apple,banana,cherry".split(",")    # ["apple",
"banana", "cherry"]
joined = "-".join(words)                    # "apple-banana-
cherry"
lines = "line1\nline2\nline3".splitlines() # ["line1", "line2",
"line3"]
```

String Slicing and Indexing

Basic Indexing

```
 text = "Python"
first_char = text[0]      # "P"
last_char = text[-1]      # "n"
second_last = text[-2]    # "o"
```

String Slicing

```
 text = "Python Programming"

# Basic slicing [start:end:step]
substring = text[0:6]       # "Python"
from_start = text[:6]       # "Python"
to_end = text[7:]           # "Programming"
every_second = text[::2]    # "Pto rgamn"
reversed_text = text[::-1] # "gnimmargorP nohtyP"

# Advanced slicing
middle = text[3:10]         # "hon Pro"
skip_chars = text[1::3]     # "yh oamn"
```

String Formatting

Old-Style Formatting (% operator)

```
 name = "Alice"
age = 25
height = 5.6
```

```
# Basic formatting
message = "Hello, %s!" % name
details = "Name: %s, Age: %d, Height: %.1f" % (name, age,
height)
```

str.format() Method

```
 name = "Bob"
age = 30
salary = 50000.75

# Positional arguments
message = "Hello, {}!".format(name)
details = "Name: {}, Age: {}, Salary: ${:.2f}".format(name, age,
salary)

# Named arguments
template = "Name: {name}, Age: {age}, Salary: ${salary:.2f}"
formatted = template.format(name=name, age=age, salary=salary)

# Index-based formatting
pattern = "{0} is {1} years old. {0} works as a {2}."
result = pattern.format("Alice", 25, "Engineer")
```

f-strings (Formatted String Literals) - Python 3.6+

```
 name = "Charlie"
age = 35
balance = 1234.567

# Basic f-string
greeting = f"Hello, {name}!"
info = f"{name} is {age} years old"

# Expressions in f-strings
calculation = f"Next year, {name} will be {age + 1}"
formatted_money = f"Balance: ${balance:.2f}"

# Format specifications
number = 42
binary = f"Binary: {number:b}"      # "Binary: 101010"
hex_val = f"Hex: {number:x}"        # "Hex: 2a"
padded = f"Padded: {number:05d}"    # "Padded: 00042"
```

Advanced String Formatting

Format Specifications

```
value = 123.456789

# Number formatting
formatted = f"{value:.2f}"      # "123.46" (2 decimal places)
scientific = f"{value:.2e}"     # "1.23e+02" (scientific
notation)
percentage = f"{0.85:.1%}"      # "85.0%" (percentage)

# Alignment and padding
text = "hello"
left_aligned = f"{text:<10}"    # "hello     "
right_aligned = f"{text:>10}"   # "     hello"
centered = f"{text:^10}"        # "  hello   "
zero_padded = f"{42:05d}"       # "00042"
```

Date and Time Formatting

```
from datetime import datetime

now = datetime.now()
formatted_date = f"{now:%Y-%m-%d %H:%M:%S}"  # "2024-12-06
14:30:45"
date_only = f"{now:%B %d, %Y}"               # "December 06,
2024"
```

String Validation and Processing

Input Validation

```
def validate_email(email):
    """Simple email validation."""
    if "@" not in email:
        return False
    if email.count("@") != 1:
        return False
    if email.startswith("@") or email.endswith("@"):
        return False
    return True

def validate_phone(phone):
    """Validate phone number format."""
    # Remove common separators
    cleaned = phone.replace("-", "").replace(" ",
```

```python
"").replace("(", "").replace(")", "")
    return cleaned.isdigit() and len(cleaned) == 10
```

Text Processing

```python
def count_words(text):
    """Count words in text."""
    words = text.split()
    return len(words)

def extract_numbers(text):
    """Extract all numbers from text."""
    numbers = []
    words = text.split()
    for word in words:
        if word.isdigit():
            numbers.append(int(word))
    return numbers

def clean_text(text):
    """Clean and normalize text."""
    # Remove extra whitespace and convert to lowercase
    cleaned = " ".join(text.split()).lower()
    # Remove punctuation (simple approach)
    punctuation = ".,!?;:"
    for char in punctuation:
        cleaned = cleaned.replace(char, "")
    return cleaned
```

Regular Expressions (Basic)

Using the re Module

```python
import re

text = "Contact us at support@example.com or sales@company.org"

# Find all email addresses
emails = re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|
a-z]{2,}\b', text)
print(emails)  # ['support@example.com', 'sales@company.org']

# Replace patterns
phone_text = "Call 123-456-7890 or 987.654.3210"
normalized = re.sub(r'[.-]', '-', phone_text)
print(normalized)  # "Call 123-456-7890 or 987-654-3210"

# Split by pattern
```

```
data = "apple,banana;cherry:orange"
fruits = re.split(r'[,;:]', data)
print(fruits)  # ['apple', 'banana', 'cherry', 'orange']
```

String Encoding and Decoding

Unicode and Encoding

```
# Unicode string
text = "Hello, 世界!"  # Contains Unicode characters

# Encoding to bytes
utf8_bytes = text.encode('utf-8')
ascii_bytes = "Hello".encode('ascii')

# Decoding from bytes
decoded_text = utf8_bytes.decode('utf-8')

# Handling encoding errors
try:
    problematic = "café".encode('ascii')
except UnicodeEncodeError:
    # Handle encoding error
    safe_encoding = "café".encode('ascii', errors='ignore')  #
"caf"
```

Common String Patterns in Exams

Password Validation

```
def validate_password(password):
    """Validate password strength."""
    if len(password) < 8:
        return False, "Password must be at least 8 characters"

    has_upper = any(c.isupper() for c in password)
    has_lower = any(c.islower() for c in password)
    has_digit = any(c.isdigit() for c in password)

    if not (has_upper and has_lower and has_digit):
        return False, "Password must contain uppercase,
lowercase, and digit"

    return True, "Password is valid"
```

Text Statistics

```python
def analyze_text(text):
    """Analyze text and return statistics."""
    words = text.split()
    sentences = text.count('.') + text.count('!') +
text.count('?')

    return {
        'characters': len(text),
        'words': len(words),
        'sentences': sentences,
        'average_word_length': sum(len(word) for word in
words) / len(words) if words else 0
    }
```

String Manipulation Challenges

```python
def reverse_words_in_sentence(sentence):
    """Reverse each word in a sentence while keeping word
order."""
    words = sentence.split()
    reversed_words = [word[::-1] for word in words]
    return " ".join(reversed_words)

def is_palindrome(text):
    """Check if text is a palindrome (ignoring case and
spaces)."""
    cleaned = text.replace(" ", "").lower()
    return cleaned == cleaned[::-1]

def title_case_custom(text):
    """Convert to title case with custom rules."""
    words = text.split()
    title_words = []
    for word in words:
        if len(word) > 3:  # Capitalize words longer than 3
characters
            title_words.append(word.capitalize())
        else:
            title_words.append(word.lower())
    return " ".join(title_words)
```

String Performance Considerations

Efficient String Building

```python
# Inefficient (creates many intermediate strings)
result = ""
for i in range(1000):
    result += str(i) + " "

# Efficient (using join)
numbers = [str(i) for i in range(1000)]
result = " ".join(numbers)

# Using list for building
parts = []
for i in range(1000):
    parts.append(str(i))
result = " ".join(parts)
```

Best Practices for String Operations

1. **Use f-strings** for string formatting in Python 3.6+
2. **Prefer str methods** over regular expressions for simple operations
3. **Use join()** for efficient string concatenation
4. **Handle encoding explicitly** when working with files or network data
5. **Validate input strings** before processing
6. **Use raw strings** for regex patterns and file paths
7. **Consider memory usage** when working with large strings
8. **Use appropriate string methods** for case-insensitive comparisons

String operations are fundamental to many programming tasks and appear frequently in exams through text processing challenges, data validation scenarios, and formatting requirements. Mastering these concepts is essential for effective Python programming.

Namespace and Scope

Understanding namespace and scope is crucial for writing correct Python programs and avoiding common errors. These concepts determine where variables can be accessed and how Python resolves variable names.

Understanding Scope

Local Scope

Variables defined inside a function have local scope and are only accessible within that function:

```python
def my_function():
    local_var = "I'm local"
    print(local_var)  # This works

my_function()
# print(local_var)  # This would cause NameError
```

Global Scope

Variables defined at the module level have global scope and can be accessed from anywhere in the module:

```python
global_var = "I'm global"

def access_global():
    print(global_var)  # Can access global variable

def another_function():
    print(global_var)  # Can also access global variable

access_global()      # Output: I'm global
another_function()   # Output: I'm global
```

The global Keyword

To modify a global variable inside a function, use the `global` keyword:

```python
counter = 0  # Global variable

def increment():
    global counter
    counter += 1  # Modify global variable

def reset():
    global counter
    counter = 0

print(counter)  # 0
increment()
```

```python
print(counter)  # 1
increment()
print(counter)  # 2
reset()
print(counter)  # 0
```

Local vs Global Variable Precedence

When a local variable has the same name as a global variable, the local variable takes precedence within the function:

```python
x = "global"

def test_scope():
    x = "local"  # This creates a new local variable
    print(f"Inside function: {x}")

test_scope()        # Output: Inside function: local
print(f"Outside function: {x}")  # Output: Outside function:
global
```

The LEGB Rule

Python follows the LEGB rule to resolve variable names: - **L**ocal: Inside the current function - **E**nclosing: In enclosing functions (for nested functions) - **G**lobal: At the module level - **B**uilt-in: In the built-in namespace

```python
# Built-in: len, print, etc.
# Global scope
global_var = "global"

def outer_function():
    # Enclosing scope
    enclosing_var = "enclosing"

    def inner_function():
        # Local scope
        local_var = "local"
        print(f"Local: {local_var}")
        print(f"Enclosing: {enclosing_var}")
        print(f"Global: {global_var}")
        print(f"Built-in: {len('hello')}")  # Built-in function

    inner_function()

outer_function()
```

Enclosing Scope and Nested Functions

Accessing Enclosing Variables

```
def outer():
    x = "outer variable"

    def inner():
        print(x)  # Can access enclosing scope

    inner()

outer()  # Output: outer variable
```

The nonlocal Keyword

To modify a variable in the enclosing scope, use the `nonlocal` keyword:

```
def outer():
    count = 0

    def increment():
        nonlocal count
        count += 1

    def get_count():
        return count

    increment()
    increment()
    print(f"Count: {get_count()}")  # Output: Count: 2

outer()
```

Closure Example

```
def create_multiplier(factor):
    def multiplier(number):
        return number * factor  # 'factor' is captured from
enclosing scope
    return multiplier

times_two = create_multiplier(2)
times_three = create_multiplier(3)
```

```python
print(times_two(5))    # 10
print(times_three(5))  # 15
```

Namespace Examples

Module Namespace

```python
 # In a file called math_utils.py
PI = 3.14159

def calculate_area(radius):
    return PI * radius ** 2

def calculate_circumference(radius):
    return 2 * PI * radius

# In another file
import math_utils

area = math_utils.calculate_area(5)
print(math_utils.PI)
```

Class Namespace

```python
 class Calculator:
    # Class variable (shared by all instances)
    precision = 2

    def __init__(self, name):
        # Instance variable (unique to each instance)
        self.name = name

    def add(self, a, b):
        # Local variables
        result = a + b
        return round(result, self.precision)

calc1 = Calculator("Basic")
calc2 = Calculator("Advanced")

# Each instance has its own namespace
print(calc1.name)  # "Basic"
print(calc2.name)  # "Advanced"

# But they share class variables
print(calc1.precision)  # 2
print(calc2.precision)  # 2
```

Common Scope-Related Errors

UnboundLocalError

```python
x = 10

def problematic_function():
    print(x)  # This will cause UnboundLocalError
    x = 20    # This makes x a local variable

# To fix this:
def fixed_function():
    global x
    print(x)  # Now this works
    x = 20
```

Modifying Mutable Objects

```python
my_list = [1, 2, 3]  # Global list

def modify_list():
    my_list.append(4)  # This works without 'global'
    print(my_list)

def reassign_list():
    global my_list
    my_list = [5, 6, 7]  # This needs 'global' to reassign

modify_list()    # [1, 2, 3, 4]
reassign_list()  # [5, 6, 7]
```

Practical Examples

Configuration Management

```python
# Global configuration
DEBUG = True
MAX_CONNECTIONS = 100

def log_message(message):
    if DEBUG:
        print(f"[DEBUG] {message}")

def get_connection_limit():
    return MAX_CONNECTIONS

def set_debug_mode(enabled):
    global DEBUG
    DEBUG = enabled

log_message("Starting application")  # Prints if DEBUG is True
set_debug_mode(False)
log_message("This won't print")      # Won't print now
```

Counter with Closure

```python
def create_counter(start=0):
    count = start

    def increment(step=1):
        nonlocal count
        count += step
        return count

    def decrement(step=1):
        nonlocal count
        count -= step
        return count

    def reset():
        nonlocal count
        count = start

    def get_value():
        return count

    # Return a dictionary of functions
    return {
        'increment': increment,
        'decrement': decrement,
        'reset': reset,
        'value': get_value
    }

counter = create_counter(10)
print(counter['value']())      # 10
print(counter['increment']())  # 11
print(counter['increment'](5)) # 16
print(counter['decrement'](3)) # 13
counter['reset']()
print(counter['value']())      # 10
```

Best Practices for Scope and Namespace

Minimize Global Variables

```python
 # Avoid this:
total = 0
count = 0

def add_number(num):
    global total, count
    total += num
    count += 1

# Prefer this:
class NumberProcessor:
    def __init__(self):
        self.total = 0
        self.count = 0

    def add_number(self, num):
        self.total += num
        self.count += 1

    def get_average(self):
        return self.total / self.count if self.count > 0 else 0
```

Use Function Parameters

```python
 # Avoid relying on global variables
def calculate_tax(amount):
    tax_rate = 0.08  # Local constant
    return amount * tax_rate

# Or pass as parameter
def calculate_tax(amount, tax_rate):
    return amount * tax_rate
```

Clear Variable Names

```python
 def process_data():
    # Use descriptive names to avoid confusion
    user_input = get_user_input()
    processed_data = clean_data(user_input)
    final_result = analyze_data(processed_data)
    return final_result
```

Debugging Scope Issues

Using locals() and globals()

```
x = "global x"

def debug_scope():
    y = "local y"
    print("Local variables:", locals())
    print("Global variables:", list(globals().keys()))

debug_scope()
```

Variable Inspection

```
def inspect_variables():
    local_var = "local"
    print(f"Local variable exists: {'local_var' in locals()}")
    print(f"Global variable exists: {'global_var' in
globals()}")

global_var = "global"
inspect_variables()
```

Common Exam Patterns

Scope Resolution Questions

```
x = 1

def func1():
    x = 2
    def func2():
        x = 3
        print(f"func2: {x}")  # What will this print?
    func2()
    print(f"func1: {x}")     # What will this print?

print(f"global: {x}")        # What will this print?
func1()
```

Global vs Local Modifications

```
count = 0
```

```python
def increment_wrong():
    count = count + 1  # UnboundLocalError!

def increment_right():
    global count
    count = count + 1

def increment_list(lst):
    lst.append(1)  # This works without global

my_list = []
increment_list(my_list)
print(my_list)  # [1]
```

Understanding scope and namespace is essential for writing correct Python programs and avoiding common errors. These concepts are frequently tested in exams through questions about variable accessibility, modification, and the behavior of nested functions.

Functions II - Advanced Concepts

unpacking

Advanced function concepts in Python include parameter properties, unpacking, recursion, and higher-order functions. These topics are essential for writing sophisticated programs and frequently appear in programming exams.

Parameter Properties and Advanced Arguments

Parameter Unpacking with *args

The `*args` parameter allows functions to accept any number of positional arguments:

```python
 def sum_all(*args):
    """Sum any number of arguments."""
    total = 0
    for num in args:
        total += num
    return total

result1 = sum_all(1, 2, 3, 4, 5)        # 15
result2 = sum_all(10, 20)               # 30
result3 = sum_all()                     # 0

# Unpacking a list into arguments
numbers = [1, 2, 3, 4, 5]
result4 = sum_all(*numbers)             # 15 (unpacks list)
```

Keyword Arguments with **kwargs

The `**kwargs` parameter allows functions to accept any number of keyword arguments:

```
def create_user(**kwargs):
    """Create a user profile from keyword arguments."""
    user = {}
    for key, value in kwargs.items():
        user[key] = value
    return user

user1 = create_user(name="Alice", age=25, city="New York")
user2 = create_user(name="Bob", email="bob@example.com",
active=True)

# Unpacking a dictionary into keyword arguments
user_data = {"name": "Charlie", "age": 30, "occupation":
"Engineer"}
user3 = create_user(**user_data)
```

Combining All Parameter Types

When defining functions with multiple parameter types, they must follow this order:

```
def complex_function(required, default_param="default", *args,
**kwargs):
    """Function demonstrating all parameter types."""
    print(f"Required: {required}")
    print(f"Default: {default_param}")
    print(f"Args: {args}")
    print(f"Kwargs: {kwargs}")

# Various ways to call this function
complex_function("value1")
complex_function("value1", "custom_default")
complex_function("value1", "custom_default", "extra1", "extra2")
complex_function("value1", extra_key="extra_value")
complex_function("value1", "custom", "arg1", "arg2",
key1="val1", key2="val2")
```

Keyword-Only Arguments

Python 3 allows defining keyword-only arguments using `*`:

```
def create_connection(host, port, *, timeout=30, ssl=False):
    """Create a connection with keyword-only parameters."""
    print(f"Connecting to {host}:{port}")
    print(f"Timeout: {timeout}, SSL: {ssl}")

# These work:
create_connection("localhost", 8080)
create_connection("localhost", 8080, timeout=60)
create_connection("localhost", 8080, ssl=True, timeout=45)

# This would cause an error:
# create_connection("localhost", 8080, 60)  # timeout must be
keyword
```

Positional-Only Arguments (Python 3.8+)

```
def calculate_area(length, width, /):
    """Calculate area with positional-only parameters."""
    return length * width

# These work:
area1 = calculate_area(10, 5)

# This would cause an error:
# area2 = calculate_area(length=10, width=5)  # Must be
positional
```

Unpacking in Function Calls

Unpacking Lists and Tuples

```
def greet(first_name, last_name, title="Mr."):
    return f"Hello, {title} {first_name} {last_name}"

# Unpacking a list
name_parts = ["John", "Doe"]
greeting1 = greet(*name_parts)  # "Hello, Mr. John Doe"

# Unpacking a tuple with additional argument
name_tuple = ("Jane", "Smith")
greeting2 = greet(*name_tuple, title="Dr.")  # "Hello, Dr. Jane
Smith"
```

Unpacking Dictionaries

```
def create_profile(name, age, city, occupation="Unknown"):
    return {
        "name": name,
        "age": age,
        "city": city,
        "occupation": occupation
    }

# Unpacking a dictionary
person_data = {
    "name": "Alice",
    "age": 30,
    "city": "Boston",
    "occupation": "Engineer"
}

profile = create_profile(**person_data)
```

Mixed Unpacking

```
def process_data(operation, *data, **options):
    """Process data with various parameters."""
    print(f"Operation: {operation}")
    print(f"Data: {data}")
    print(f"Options: {options}")

# Mixed unpacking
numbers = [1, 2, 3, 4, 5]
settings = {"verbose": True, "format": "json"}

process_data("sum", *numbers, **settings)
```

Recursion

recursive

Basic Recursion Concepts

Recursion is a programming technique where a function calls itself to solve a problem by breaking it down into smaller, similar subproblems:

```
def factorial(n):
    """Calculate factorial using recursion."""
    # Base case
    if n <= 1:
        return 1
    # Recursive case
    else:
        return n * factorial(n - 1)

result = factorial(5)  # 5 * 4 * 3 * 2 * 1 = 120
```

Fibonacci Sequence

```python
def fibonacci(n):
    """Calculate nth Fibonacci number."""
    # Base cases
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    # Recursive case
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# Generate first 10 Fibonacci numbers
fib_sequence = [fibonacci(i) for i in range(10)]
print(fib_sequence)  # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Optimized Fibonacci with Memoization

```python
def fibonacci_memo(n, memo={}):
    """Optimized Fibonacci with memoization."""
    if n in memo:
        return memo[n]

    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        memo[n] = fibonacci_memo(n - 1, memo) + fibonacci_memo(n
- 2, memo)
        return memo[n]

# Much faster for large numbers
result = fibonacci_memo(50)
```

Tree Traversal

```python
def sum_tree(node):
    """Sum all values in a binary tree."""
    if node is None:
        return 0

    # Sum current node + left subtree + right subtree
    return node.value + sum_tree(node.left) +
sum_tree(node.right)

def find_max_depth(node):
    """Find maximum depth of a binary tree."""
    if node is None:
        return 0

    left_depth = find_max_depth(node.left)
    right_depth = find_max_depth(node.right)

    return 1 + max(left_depth, right_depth)
```

List Processing with Recursion

```python
def sum_list(lst):
    """Sum list elements using recursion."""
    if not lst:  # Empty list
        return 0
    else:
        return lst[0] + sum_list(lst[1:])

def reverse_list(lst):
    """Reverse a list using recursion."""
    if len(lst) <= 1:
        return lst
    else:
        return [lst[-1]] + reverse_list(lst[:-1])

def flatten_list(nested_list):
    """Flatten a nested list structure."""
    result = []
    for item in nested_list:
        if isinstance(item, list):
            result.extend(flatten_list(item))
        else:
            result.append(item)
    return result

# Example usage
nested = [1, [2, 3], [4, [5, 6]], 7]
flat = flatten_list(nested)  # [1, 2, 3, 4, 5, 6, 7]
```

Higher-Order Functions

Functions as Arguments

```python
def apply_operation(numbers, operation):
    """Apply an operation to all numbers in a list."""
    result = []
    for num in numbers:
        result.append(operation(num))
    return result

def square(x):
    return x ** 2

def cube(x):
    return x ** 3

numbers = [1, 2, 3, 4, 5]
squared = apply_operation(numbers, square)   # [1, 4, 9, 16, 25]
cubed = apply_operation(numbers, cube)       # [1, 8, 27, 64,
125]
```

Functions Returning Functions

```
 def create_multiplier(factor):
    """Create a function that multiplies by a given factor."""
    def multiplier(x):
        return x * factor
    return multiplier

# Create specific multiplier functions
double = create_multiplier(2)
triple = create_multiplier(3)

print(double(5))  # 10
print(triple(5))  # 15

def create_validator(min_length, max_length):
    """Create a string length validator."""
    def validate(text):
        length = len(text)
        return min_length <= length <= max_length
    return validate

# Create specific validators
username_validator = create_validator(3, 20)
password_validator = create_validator(8, 50)

print(username_validator("abc"))      # True
print(password_validator("short"))    # False
```

Decorators (Basic Introduction)

```
 def timing_decorator(func):
    """Decorator to measure function execution time."""
    import time

    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} took {end_time - start_time:.
4f} seconds")
        return result

    return wrapper

@timing_decorator
def slow_function():
    """A function that takes some time."""
    import time
    time.sleep(1)
    return "Done"

# When called, it will print timing information
result = slow_function()
```

Built-in Higher-Order Functions

map() Function

```
 # Apply a function to all items in an iterable
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))  # [1, 4, 9, 16,
25]

# Map with multiple iterables
list1 = [1, 2, 3]
list2 = [4, 5, 6]
sums = list(map(lambda x, y: x + y, list1, list2))  # [5, 7, 9]
```

filter() Function

```
 # Filter items based on a condition
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evens = list(filter(lambda x: x % 2 == 0, numbers))
# [2, 4, 6, 8, 10]

# Filter strings by length
words = ["apple", "hi", "banana", "cat", "elephant"]
long_words = list(filter(lambda word: len(word) > 4, words))  #
["apple", "banana", "elephant"]
```

reduce() Function

```python
from functools import reduce

# Reduce a list to a single value
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)  # 120 (1*2*3*4*5)

# Find maximum value
maximum = reduce(lambda x, y: x if x > y else y, numbers)  # 5

# Concatenate strings
words = ["Hello", " ", "World", "!"]
sentence = reduce(lambda x, y: x + y, words)  # "Hello World!"
```

Function Annotations and Type Hints

Basic Type Annotations

```python
def add_numbers(a: int, b: int) -> int:
    """Add two integers and return the result."""
    return a + b

def greet_user(name: str, age: int = 25) -> str:
    """Greet a user with their name and age."""
    return f"Hello {name}, you are {age} years old"

def process_items(items: list[str]) -> dict[str, int]:
    """Count the length of each item."""
    return {item: len(item) for item in items}
```

Complex Type Annotations

```python
from typing import List, Dict, Optional, Union, Callable

def process_data(
    data: List[Dict[str, Union[str, int]]],
    processor: Callable[[Dict], str]
) -> Optional[List[str]]:
    """Process a list of dictionaries using a processor
function."""
    if not data:
        return None

    return [processor(item) for item in data]
```

Advanced Function Patterns

Function Factories

```python
def create_calculator(operation):
    """Create a calculator function for a specific operation."""
    operations = {
        'add': lambda x, y: x + y,
        'subtract': lambda x, y: x - y,
        'multiply': lambda x, y: x * y,
        'divide': lambda x, y: x / y if y != 0 else None
    }

    return operations.get(operation, lambda x, y: None)

# Create specific calculators
adder = create_calculator('add')
multiplier = create_calculator('multiply')

print(adder(5, 3))      # 8
print(multiplier(4, 6)) # 24
```

Partial Function Application

```python
from functools import partial

def power(base, exponent):
    """Calculate base raised to exponent."""
    return base ** exponent

# Create specialized functions
square = partial(power, exponent=2)
cube = partial(power, exponent=3)

print(square(5))  # 25 (5^2)
print(cube(3))    # 27 (3^3)
```

Best Practices for Advanced Functions

1. **Use clear parameter names** and type annotations
2. **Document complex functions** with detailed docstrings
3. **Avoid deep recursion** without optimization (Python has a recursion limit)
4. **Use memoization** for expensive recursive calculations
5. **Prefer iterative solutions** when recursion isn't naturally suited
6. **Use unpacking judiciously** - it can make code less readable if overused
7. **Consider performance implications** of higher-order functions

8. **Use built-in functions** (map, filter, reduce) when appropriate

Advanced function concepts are essential for writing sophisticated Python programs and frequently appear in exams through recursion problems, parameter manipulation challenges, and functional programming scenarios.

Object-Oriented Programming I

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into classes and objects. Understanding OOP concepts is essential for writing maintainable, scalable Python programs and is frequently tested in programming exams.

Classes and Objects

Basic Class Definition

A class is a blueprint for creating objects. It defines attributes (data) and methods (functions) that objects of the class will have:

```python
class Car:
    """A simple car class."""

    def __init__(self, make, model, year):
        """Initialize a car instance."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer = 0

    def start_engine(self):
        """Start the car's engine."""
        print(f"The {self.year} {self.make} {self.model} engine
is now running.")

    def drive(self, miles):
        """Drive the car and update odometer."""
        self.odometer += miles
        print(f"Drove {miles} miles. Total: {self.odometer}
miles.")

# Creating objects (instances)
my_car = Car("Toyota", "Camry", 2020)
your_car = Car("Honda", "Civic", 2019)

# Using object methods
```

```
my_car.start_engine()
my_car.drive(100)
```

The init Method (Constructor)

The `__init__` method is called when an object is created. It initializes the object's attributes:

```
class Student:
    def __init__(self, name, student_id, grade=9):
        """Initialize a student with name, ID, and optional
grade."""
        self.name = name
        self.student_id = student_id
        self.grade = grade
        self.courses = []  # Empty list for courses

    def enroll_course(self, course):
        """Enroll student in a course."""
        self.courses.append(course)
        print(f"{self.name} enrolled in {course}")

    def get_info(self):
        """Return student information."""
        return f"Student: {self.name}, ID: {self.student_id},
Grade: {self.grade}"

# Creating student objects
alice = Student("Alice Johnson", "S001", 10)
bob = Student("Bob Smith", "S002")  # Uses default grade

print(alice.get_info())
alice.enroll_course("Mathematics")
```

Class Attributes vs Instance Attributes

Class Attributes

Class attributes are shared by all instances of the class:

```
class BankAccount:
    # Class attributes
    bank_name = "Python Bank"
    interest_rate = 0.02
    total_accounts = 0

    def __init__(self, account_holder, initial_balance=0):
        # Instance attributes
        self.account_holder = account_holder
        self.balance = initial_balance
        self.account_number = f"ACC{BankAccount.total_accounts
+ 1:04d}"

        # Update class attribute
        BankAccount.total_accounts += 1

    def deposit(self, amount):
        """Deposit money to the account."""
        self.balance += amount
        print(f"Deposited ${amount}. New balance: $
{self.balance}")

    def get_account_info(self):
        """Get account information."""
        return f"Account: {self.account_number}, Holder:
{self.account_holder}, Balance: ${self.balance}"

# Creating accounts
account1 = BankAccount("Alice", 1000)
account2 = BankAccount("Bob", 500)

print(f"Bank: {BankAccount.bank_name}")
print(f"Total accounts: {BankAccount.total_accounts}")
print(account1.get_account_info())
print(account2.get_account_info())
```

Accessing and Modifying Attributes

```
class Circle:
    pi = 3.14159  # Class attribute

    def __init__(self, radius):
        self.radius = radius  # Instance attribute

    def area(self):
        return Circle.pi * self.radius ** 2

    def circumference(self):
        return 2 * Circle.pi * self.radius

circle1 = Circle(5)
circle2 = Circle(3)

# Accessing class attribute through class
print(f"Pi value: {Circle.pi}")

# Accessing class attribute through instance
```

```python
print(f"Pi through instance: {circle1.pi}")

# Modifying class attribute affects all instances
Circle.pi = 3.14
print(f"Circle1 area: {circle1.area()}")
print(f"Circle2 area: {circle2.area()}")
```

Methods

Instance Methods

Instance methods operate on individual objects and have access to instance attributes:

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        """Calculate area of the rectangle."""
        return self.width * self.height

    def perimeter(self):
        """Calculate perimeter of the rectangle."""
        return 2 * (self.width + self.height)

    def scale(self, factor):
        """Scale the rectangle by a factor."""
        self.width *= factor
        self.height *= factor

    def is_square(self):
        """Check if the rectangle is a square."""
        return self.width == self.height

rect = Rectangle(4, 6)
print(f"Area: {rect.area()}")
print(f"Is square: {rect.is_square()}")
rect.scale(2)
print(f"New area after scaling: {rect.area()}")
```

Class Methods

Class methods operate on the class itself and are decorated with `@classmethod`:

```python
class Person:
    population = 0
    def __init__(self, name, age):
        self.name = name
        self.age = age
        Person.population += 1

    @classmethod
    def get_population(cls):
        """Get total population count."""
        return cls.population

    @classmethod
    def create_baby(cls, name):
        """Create a baby (age 0)."""
        return cls(name, 0)

    @classmethod
    def from_birth_year(cls, name, birth_year):
        """Create person from birth year."""
        import datetime
        current_year = datetime.datetime.now().year
        age = current_year - birth_year
        return cls(name, age)

# Using class methods
baby = Person.create_baby("Charlie")
adult = Person.from_birth_year("Diana", 1990)

print(f"Population: {Person.get_population()}")
print(f"Baby age: {baby.age}")
print(f"Adult age: {adult.age}")
```

Static Methods

Static methods don't access class or instance data and are decorated with `@staticmethod`:

```
class MathUtils:
    @staticmethod
    def add(a, b):
        """Add two numbers."""
        return a + b

    @staticmethod
    def is_even(number):
        """Check if a number is even."""
        return number % 2 == 0

    @staticmethod
    def factorial(n):
        """Calculate factorial of n."""
        if n <= 1:
            return 1
        result = 1
        for i in range(2, n + 1):
            result *= i
        return result

# Using static methods (can be called on class or instance)
result = MathUtils.add(5, 3)
is_even = MathUtils.is_even(10)
fact = MathUtils.factorial(5)

print(f"5 + 3 = {result}")
print(f"10 is even: {is_even}")
print(f"5! = {fact}")
```

Properties

Properties allow you to define methods that can be accessed like attributes:

```
class Temperature:
    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
        """Get temperature in Celsius."""
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        """Set temperature in Celsius."""
        if value < -273.15:
            raise ValueError("Temperature cannot be below
absolute zero")
        self._celsius = value

    @property
    def fahrenheit(self):
        """Get temperature in Fahrenheit."""
        return (self._celsius * 9/5) + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        """Set temperature using Fahrenheit."""
        self.celsius = (value - 32) * 5/9

    @property
    def kelvin(self):
        """Get temperature in Kelvin."""
        return self._celsius + 273.15

# Using properties
temp = Temperature(25)
print(f"Celsius: {temp.celsius}")
print(f"Fahrenheit: {temp.fahrenheit}")
print(f"Kelvin: {temp.kelvin}")

# Setting temperature using different scales
temp.fahrenheit = 100
print(f"After setting to 100°F: {temp.celsius}°C")
```

Special Methods (Magic Methods)

Special methods allow objects to work with built-in functions and operators:

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        """String representation for users."""
        return f"'{self.title}' by {self.author}"

    def __repr__(self):
        """String representation for developers."""
        return f"Book('{self.title}', '{self.author}',
{self.pages})"

    def __len__(self):
        """Return number of pages."""
        return self.pages

    def __eq__(self, other):
        """Check equality based on title and author."""
        if isinstance(other, Book):
            return self.title == other.title and self.author ==
other.author
        return False

    def __lt__(self, other):
        """Compare books by number of pages."""
        if isinstance(other, Book):
            return self.pages < other.pages
        return NotImplemented

# Using special methods
book1 = Book("1984", "George Orwell", 328)
```

```
book2 = Book("Animal Farm", "George Orwell", 112)

print(str(book1))         # Uses __str__
print(repr(book1))        # Uses __repr__
print(len(book1))         # Uses __len__
print(book1 == book2)     # Uses __eq__
print(book1 > book2)      # Uses __lt__ (reversed)
```

Encapsulation and Data Hiding

Private Attributes

Python uses naming conventions to indicate private attributes:

```
 class BankAccount:
    def __init__(self, account_number, initial_balance):
        self.account_number = account_number
        self._balance = initial_balance  # Protected
(convention)
        self.__pin = "1234"             # Private (name
mangling)

    def get_balance(self):
        """Get account balance."""
        return self._balance

    def deposit(self, amount, pin):
        """Deposit money with PIN verification."""
        if self.__verify_pin(pin):
            self._balance += amount
            return True
        return False

    def __verify_pin(self, pin):
        """Private method to verify PIN."""
        return pin == self.__pin

    def change_pin(self, old_pin, new_pin):
        """Change account PIN."""
        if self.__verify_pin(old_pin):
            self.__pin = new_pin
            return True
        return False

account = BankAccount("12345", 1000)
print(f"Balance: ${account.get_balance()}")

# This works (protected attribute - convention only)
print(f"Direct balance access: ${account._balance}")
```

```
# This doesn't work (private attribute with name mangling)
# print(account.__pin)  # AttributeError

# But this works (name mangling makes it _ClassName__attribute)
print(f"Mangled PIN access: {account._BankAccount__pin}")
```

Class Design Patterns

Builder Pattern

```
class Pizza:
    def __init__(self):
        self.size = None
        self.crust = None
        self.toppings = []

    def __str__(self):
        return f"{self.size} {self.crust} pizza with {',
'.join(self.toppings)}"

class PizzaBuilder:
    def __init__(self):
        self.pizza = Pizza()

    def set_size(self, size):
        self.pizza.size = size
        return self  # Return self for method chaining

    def set_crust(self, crust):
        self.pizza.crust = crust
        return self

    def add_topping(self, topping):
        self.pizza.toppings.append(topping)
        return self

    def build(self):
        return self.pizza

# Using the builder pattern
pizza = (PizzaBuilder()
         .set_size("Large")
         .set_crust("Thin")
         .add_topping("Pepperoni")
         .add_topping("Mushrooms")
         .add_topping("Cheese")
         .build())

print(pizza)
```

Singleton Pattern

```python
class DatabaseConnection:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance.connected = False
        return cls._instance

    def connect(self):
        if not self.connected:
            print("Connecting to database...")
            self.connected = True
        else:
            print("Already connected to database")

    def disconnect(self):
        if self.connected:
            print("Disconnecting from database...")
            self.connected = False

# Testing singleton behavior
db1 = DatabaseConnection()
db2 = DatabaseConnection()

print(f"Same instance: {db1 is db2}")  # True
db1.connect()
db2.connect()  # Already connected message
```

Common OOP Patterns in Exams

Counter Class

```python
class Counter:
    def __init__(self, start=0):
        self._count = start

    def increment(self, step=1):
        self._count += step

    def decrement(self, step=1):
        self._count -= step

    def reset(self):
        self._count = 0

    @property
    def value(self):
        return self._count

    def __str__(self):
        return f"Counter: {self._count}"

counter = Counter(10)
counter.increment(5)
print(counter.value)  # 15
```

Student Grade Management

```python
class Student:
    def __init__(self, name, student_id):
        self.name = name
        self.student_id = student_id
        self._grades = {}

    def add_grade(self, subject, grade):
        if 0 <= grade <= 100:
            self._grades[subject] = grade
        else:
            raise ValueError("Grade must be between 0 and 100")

    def get_average(self):
        if not self._grades:
            return 0
        return sum(self._grades.values()) / len(self._grades)

    def get_letter_grade(self):
        avg = self.get_average()
        if avg >= 90: return 'A'
        elif avg >= 80: return 'B'
        elif avg >= 70: return 'C'
        elif avg >= 60: return 'D'
        else: return 'F'

student = Student("Alice", "S001")
student.add_grade("Math", 85)
student.add_grade("Science", 92)
print(f"Average: {student.get_average():.1f}")
print(f"Letter grade: {student.get_letter_grade()}")
```

Best Practices for OOP

1. **Use clear, descriptive class names** (PascalCase)
2. **Keep classes focused** on a single responsibility
3. **Use properties** for controlled attribute access

4. **Implement appropriate special methods** for natural object behavior
5. **Follow encapsulation principles** - hide internal implementation details
6. **Use class methods** for alternative constructors
7. **Use static methods** for utility functions related to the class
8. **Document classes and methods** with clear docstrings

Object-oriented programming is fundamental to Python and frequently tested in exams through class design problems, inheritance scenarios, and method implementation challenges. Understanding these concepts is essential for writing maintainable and scalable code.

Quick Reference Guide

Essential Python Syntax

Variable Assignment and Basic Operations

```
# Variables
name = "Alice"
age = 25
height = 5.6
is_student = True

# Basic operations
result = 10 + 5 * 2  # 20 (multiplication first)
remainder = 17 % 5   # 2 (modulo)
power = 2 ** 3       # 8 (exponentiation)
```

Control Structures Quick Reference

```
# If statements
if condition:
    # code
elif other_condition:
    # code
else:
    # code

# For loops
for item in iterable:
    # code

for i in range(start, stop, step):
    # code
```

```python
# While loops
while condition:
    # code
```

Function Definition Template

```python
def function_name(param1, param2="default", *args, **kwargs):
    """
    Function description.

    Args:
        param1: Description
        param2: Description with default
        *args: Variable positional arguments
        **kwargs: Variable keyword arguments

    Returns:
        Description of return value
    """
    # Function body
    return result
```

Class Definition Template

```python
class ClassName:
    # Class variables
    class_var = "shared"

    def __init__(self, param1, param2):
        """Initialize instance."""
        self.param1 = param1
        self.param2 = param2

    def instance_method(self):
        """Instance method."""
        return self.param1

    @classmethod
    def class_method(cls):
        """Class method."""
        return cls.class_var

    @staticmethod
    def static_method():
        """Static method."""
        return "utility function"

    @property
    def computed_property(self):
        """Property getter."""
        return self.param1 + self.param2
```

Common Patterns and Idioms

List Comprehensions

```python
# Basic pattern: [expression for item in iterable if condition]
squares = [x**2 for x in range(10)]
evens = [x for x in range(20) if x % 2 == 0]
words_upper = [word.upper() for word in words if len(word) > 3]

# Nested comprehensions
matrix = [[i*j for j in range(3)] for i in range(3)]
```

Dictionary Comprehensions

```python
# Pattern: {key_expr: value_expr for item in iterable if
condition}
squares_dict = {x: x**2 for x in range(5)}
word_lengths = {word: len(word) for word in words}
```

Exception Handling

```python
try:
    # risky code
    result = operation()
except SpecificError as e:
    # handle specific error
    print(f"Error: {e}")
except (Error1, Error2):
    # handle multiple error types
    print("Multiple error types")
except Exception as e:
    # handle any other error
    print(f"Unexpected error: {e}")
else:
    # runs if no exception
    print("Success")
finally:
    # always runs
    cleanup()
```

File Operations Quick Reference

```
 # Reading files
with open("file.txt", "r") as f:
    content = f.read()          # Read entire file
    lines = f.readlines()       # Read all lines as list
    for line in f:              # Iterate line by line
        process(line.strip())

# Writing files
with open("file.txt", "w") as f:
    f.write("content")          # Write string
    f.writelines(lines)         # Write list of strings

# CSV operations
import csv
with open("data.csv", "r") as f:
    reader = csv.reader(f)
    for row in reader:
        process(row)
```

String Operations Summary

```
 # Common string methods
text.upper()           # Convert to uppercase
text.lower()           # Convert to lowercase
text.strip()           # Remove whitespace
text.split(",")        # Split by delimiter
",".join(list)         # Join list with delimiter
text.replace("old", "new")  # Replace substring
text.find("substring") # Find position (-1 if not found)
text.startswith("pre") # Check prefix
text.endswith("suf")   # Check suffix

# String formatting
f"Hello {name}, you are {age} years old"
"Hello {}, you are {} years old".format(name, age)
"Hello {name}, you are {age} years old".format(name=name,
age=age)
```

Common Built-in Functions

```
 # Type conversion
int("123")        # String to integer
float("3.14")     # String to float
str(42)           # Number to string
list("hello")     # String to list of characters
```

```
tuple([1,2,3])    # List to tuple

# Sequence operations
len(sequence)     # Length
max(sequence)     # Maximum value
min(sequence)     # Minimum value
sum(numbers)      # Sum of numbers
sorted(sequence)  # Return sorted copy
reversed(sequence) # Return reversed iterator

# Functional programming
map(function, iterable)      # Apply function to all items
filter(function, iterable)   # Filter items by condition
zip(iter1, iter2)           # Combine iterables
enumerate(iterable)         # Add indices
```

Debugging Tips for Exams

1. **Check indentation** - Python is sensitive to whitespace
2. **Verify variable names** - case-sensitive and no typos
3. **Match parentheses and brackets** - ensure proper nesting
4. **Check function calls** - correct number and order of arguments
5. **Validate logic flow** - trace through conditions and loops
6. **Test edge cases** - empty lists, zero values, boundary conditions
7. **Review scope** - ensure variables are accessible where used
8. **Check return statements** - functions should return appropriate values

Common Exam Mistakes to Avoid

1. **Forgetting `self` parameter** in instance methods
2. **Using `=` instead of `==`** for comparison
3. **Modifying lists while iterating** over them
4. **Forgetting to return values** from functions
5. **Incorrect indentation** in nested structures
6. **Using mutable default arguments** in functions
7. **Confusing class vs instance variables**
8. **Not handling exceptions** when required
9. **Forgetting to close files** (use `with` statement)
10. **Off-by-one errors** in loops and indexing

Conclusion

This comprehensive Python exam summary covers all essential topics for pen-and-paper examinations. The material is organized to facilitate quick reference during exams while providing detailed explanations and practical examples for thorough understanding.

Key areas to focus on for exam success: - **Syntax mastery**: Ensure perfect understanding of Python syntax rules - **Problem-solving patterns**: Practice common algorithmic approaches - **Object-oriented concepts**: Understand classes, inheritance, and encapsulation - **Error handling**: Know when and how to use exception handling - **Code organization**: Write clean, readable, and well-structured code

Regular practice with the provided examples and patterns will build confidence and competency in Python programming. Remember to always test your logic mentally and consider edge cases when solving problems.

Good luck with your Python programming examination!