Python Midterm Study Guide

Table of Contents

1. Basic Output

2. Variables

3. Programming Errors

4. Arithmetic Operations

5. Comments and Documentation

6. Input and Type Casting

7. Math Module

8. Floating Point Numbers

9. Conditional Statements

10.Strings

11.Loops

1. Basic Output

Concept Overview

The `print()` function is used to display output to the computer screen. It's one of the most fundamental functions in Python.

Key Points

1. Basic syntax: print("Hello")

2. Multiple items: print("Hello", "World")

3. Special arguments: sep and end

4. String repetition with *

Examples

```
# Basic print
print("Hello")  # Output: Hello

# Multiple items
print("Hello", "World")  # Output: Hello World

# Using sep
```

```
print("Hello", "World", sep="-")  # Output: Hello-World

# Using end
print("Hello", end="!")  # Output: Hello!

# String repetition
print("-" * 10)  # Output: ----------
```

Common Mistakes

1. ❌ Forgetting parentheses

```
print "Hello"  # SyntaxError
```

2. ❌ Using commas instead of + for string concatenation in a single argument

```
print("Hello", "World")  # Prints with space

print("Hello" + "World")  # Prints without space
```

Practice Problems

1. Write a program to print your name and age on separate lines.

2. Print the following pattern:

   *

   **

   ***

3. Print "Hello" and "World" with three dashes in between.

Solutions

```
# 1. Name and age

print("John Doe")

print(25)


# 2. Pattern

print("*")

print("*" * 2)

print("*" * 3)


# 3. Hello-World

print("Hello", "World", sep="---")
```

2. Variables

Concept Overview

Variables are containers for storing data values. Python has no command for declaring variables - they are created when first assigned a value.

Key Points

1. Variables are dynamically typed
2. Naming conventions and rules
3. Multiple assignment
4. Different data types (int, float, string)

Examples

```
# Basic assignment
name = "John"
age = 25
height = 1.75


# Multiple assignment
x, y, z = 1, 2, 3


# Same value to multiple variables
a = b = c = 0


# Dynamic typing
x = 5      # x is an integer
x = "John"  # x is now a string
```

Common Mistakes

1. ❌ Using invalid variable names

```
2name = "John"    # Can't start with number
my-name = "John"  # Can't use hyphen
```

2. ❌ Using Python keywords as variable names

```
if = 5  # 'if' is a keyword
```

Practice Problems

1. Create variables for your first name, last name, and age. Then print them.

2. Swap the values of two variables without using a third variable.

3. Assign the same value to three different variables in one line.

Solutions

```
# 1. Personal info
first_name = "John"
last_name = "Doe"
age = 25
print(first_name, last_name, age)


# 2. Swap variables
x = 5
y = 10
x, y = y, x  # x is now 10, y is now 5


# 3. Same value to multiple variables
a = b = c = 100
```

3. Programming Errors

Concept Overview

Understanding different types of errors is crucial for debugging and writing correct code.

Types of Errors

1. Syntax Errors

- Errors in the structure of the code
- Detected before the program runs

2. Runtime Errors

- Occur while the program is running
- Often due to invalid input or operations

3. Logic Errors

- Program runs but produces incorrect results
- Most difficult to find and fix

Examples

```
# 1. Syntax Error
print("Hello"  # Missing closing parenthesis


# 2. Runtime Error
x = 5
y = 0
print(x / y)  # Division by zero


# 3. Logic Error
def calculate_average(a, b):
   return a + b  # Should be (a + b) / 2
```

Common Mistakes and How to Fix Them

1. Indentation Errors

```
# Wrong
if x > 0:
print("Positive")  # IndentationError


# Correct
if x > 0:
   print("Positive")
```

2. Type Errors

```
# Wrong
x = "5"
y = 2
print(x + y)  # TypeError
# Correct
x = int("5")
```

```
y = 2
print(x + y)
```

Practice Problems

1. Find and fix the errors in the following code:

```
def calculate_sum(a b):
    return a + b

print(calculate_sum(3, 4))
```

2. What's wrong with this code? How would you fix it?

```
age = input("Enter your age: ")
if age > 18:
    print("You are an adult")
```

Solutions

1. Missing comma in function parameters

```
def calculate_sum(a, b):
    return a + b
print(calculate_sum(3, 4))
# 2. input() returns string, need to convert to int
age = int(input("Enter your age: "))
if age > 18:
    print("You are an adult")
```

4. Arithmetic Operations

Concept Overview

Python supports various arithmetic operators for mathematical operations.

Key Operators

1. Addition: `+`
2. Subtraction: `-`
3. Multiplication: `*`
4. Division: `/`
5. Floor Division: `//`
6. Modulus: `%`
7. Exponentiation: `**`

Examples

```
# Basic arithmetic
print(5 + 3)   # 8
print(5 - 3)   # 2
print(5 * 3)   # 15
print(5 / 3)   # 1.6666666666666667
print(5 // 3)  # 1
print(5 % 3)   # 2
print(5 ** 3)  # 125


# Order of operations
print(5 + 3 * 2)    # 11
print((5 + 3) * 2)  # 16
```

Assignment Operators

```
x = 5
x += 3   # Same as x = x + 3
x -= 2   # Same as x = x - 2
x *= 4   # Same as x = x * 4
x /= 2   # Same as x = x / 2
```

Common Mistakes

1. ❌ Integer division in Python 3

```
x = 5 / 2   # Returns 2.5, not 2
x = 5 // 2  # Returns 2 (floor division)
```

Practice Problems

1. Write a program to convert temperature from Fahrenheit to Celsius.

   Formula: C = (F - 32) * 5/9

2. Calculate the area and perimeter of a rectangle given its length and width.

3. Determine if a number is odd or even using the modulus operator.

```
# 1. Temperature conversion
fahrenheit = 98.6
celsius = (fahrenheit - 32) * 5/9
print(f"{fahrenheit}°F is {celsius:.2f}°C")

# 2. Rectangle calculations
length = 5
width = 3
area = length * width
perimeter = 2 * (length + width)
print(f"Area: {area}, Perimeter: {perimeter}")

# 3. Odd or even
number = 7
if number % 2 == 0:
    print("Even")
else:
    print("Odd")
```

5. Comments and Documentation

Concept Overview

Comments are used to explain code and make it more readable. They are ignored by the Python interpreter.

Types of Comments

1. Single-line comments: Start with `#`

2. Multi-line comments: Enclosed in triple quotes `"""`

Examples

```
# This is a single-line comment


"""
This is a multi-line comment
It can span multiple lines
Useful for longer explanations
"""


def calculate_area(radius):
    """
    This function calculates the area of a circle
    Parameters:
        radius (float): The radius of the circle
    Returns:
        float: The area of the circle
    """
    return 3.14 * radius ** 2
```

Best Practices

1. Write meaningful comments

2. Update comments when code changes

3. Don't state the obvious

Common Mistakes

1. ❌ Over-commenting obvious code

```
# Assigning 5 to x
x = 5  # Unnecessary comment
```

2. ❌ Not updating comments when code changes

```
# Calculates area of rectangle
def calculate_area(radius):  # Comment is now incorrect
    return 3.14 * radius ** 2
```

6. Input and Type Casting

Concept Overview

Python uses the `input()` function to accept user input and provides various functions for type casting (converting between data types).

Key Points

1. input() always returns a string

2. Common type casting functions:

   - int(): Convert to integer

   - float(): Convert to float

   - str(): Convert to string

Examples

```
# Basic input
name = input("Enter your name: ")
print(f"Hello, {name}!")


# Type casting input
age_str = input("Enter your age: ")
age = int(age_str)


# Direct type casting
height = float(input("Enter your height in meters: "))


# Multiple inputs
```

```
x, y = input("Enter two numbers separated by space: ").split()
x, y = int(x), int(y)
```

Common Mistakes

1. ❌ Forgetting to type cast numeric input

```
age = input("Enter age: ")
drinking_age = 21
if age >= drinking_age:  # TypeError: string comparison
```

2. ❌ Invalid type casting

```
number = int("12.5")  # ValueError: invalid literal
```

Practice Problems

1. Write a program that asks for a person's birth year and calculates their age.
2. Create a simple calculator that takes two numbers and an operator (+, -, *, /) as input.
3. Write a program that converts temperature from Celsius to Fahrenheit using user input.

Solutions

```
# 1. Age calculator
import datetime
birth_year = int(input("Enter your birth year: "))
current_year = datetime.datetime.now().year
age = current_year - birth_year
print(f"You are approximately {age} years old")


# 2. Simple calculator
num1 = float(input("Enter first number: "))
operator = input("Enter operator (+,-,*,/): ")
num2 = float(input("Enter second number: "))


if operator == '+':
   result = num1 + num2
elif operator == '-':
   result = num1 - num2
```

```
elif operator == '*':
    result = num1 * num2
elif operator == '/':
    result = num1 / num2 if num2 != 0 else "Error: Division by zero"
else:
    result = "Invalid operator"

print(f"Result: {result}")

# 3. Temperature converter
celsius = float(input("Enter temperature in Celsius: "))
fahrenheit = (celsius * 9/5) + 32
print(f"{celsius}°C is {fahrenheit}°F")
```

7. Math Module

Concept Overview

Python's math module provides access to mathematical functions and constants.

Key Functions and Constants

1. Constants: `math.pi`, `math.e'

2. Functions:

- `math.sqrt()`: Square root

- `math.ceil()`: Round up

- `math.floor()`: Round down

- `math.pow()`: Power function

- `math.sin()`, `math.cos()`, `math.tan()`: Trigonometric functions

Examples

```
import math


# Constants
print(math.pi)  # 3.141592653589793
print(math.e)  # 2.718281828459045


# Functions
print(math.sqrt(16))    # 4.0
print(math.ceil(4.2))   # 5
print(math.floor(4.8))  # 4
print(math.pow(2, 3))   # 8.0


# Trigonometry
print(math.sin(math.pi/2))  # 1.0
print(math.cos(0))        # 1.0
```

Common Mistakes

1. ❌ Forgetting to import the math module

```
print(sqrt(16))  # NameError: name 'sqrt' is not defined
```

2. ❌ Using math functions with inappropriate types

```
import math

math.sqrt(-1)  # ValueError: math domain error
```

Practice Problems

1. Write a program to calculate the area of a circle using math.pi.

3. Write a program that rounds a number both up and down and shows the difference.

Solutions

```
import math

# 1. Circle area

def circle_area(radius):

   return math.pi * math.pow(radius, 2)

print(f"Area of circle with radius 5: {circle_area(5):.2f}")

# 3. Rounding comparison

number = 3.7

print(f"Original number: {number}")

print(f"Rounded up: {math.ceil(number)}")

print(f"Rounded down: {math.floor(number)}")

print(f"Difference: {math.ceil(number) - math.floor(number)}")
```

8. Floating Point Numbers

Concept Overview

Floating-point numbers represent real numbers in Python, but they can sometimes behave in unexpected ways due to how computers represent decimals.

Key Points

1. Precision limitations
2. Scientific notation
3. Rounding functions
4. Avoiding floating-point equality comparison

Examples

```
# Basic floating-point
x = 0.1 + 0.2
print(x)  # 0.30000000000000004
# Scientific notation
large_num = 1e6   # 1 million
small_num = 1e-6  # 0.000001
# Rounding
print(round(3.14159, 2))   # 3.14
print(f"{3.14159:.2f}")    # 3.14
# Avoiding equality comparison
x = 0.1 + 0.2
y = 0.3
print(abs(x - y) < 1e-9)   # True
```

Common Mistakes

1. ❌ Direct equality comparison

```
x = 0.1 + 0.2
print(x == 0.3)  # False! Use abs(x - 0.3) < 1e-9 instead
```

9. Conditional Statements

Concept Overview

Conditional statements allow the program to make decisions and execute different code based on different conditions.

Types of Conditional Statements

1. if statement
2. if-else statement
3. if-elif-else statement
4. Nested if statements

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


# if-else
temperature = 25
if temperature > 30:
    print("It's hot!")
else:
    print("It's not hot")


# if-elif-else
score = 85
if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
elif score >= 70:
    grade = 'C'
else:
    grade = 'F'
```

```
# Nested if
num = 5
if num > 0:
  if num < 10:
    print("Single digit positive number")
```

Comparison Operators

- Equal to: `==`
- Not equal to: `!=`
- Greater than: `>`
- Less than: `<`
- Greater than or equal to: `>=`
- Less than or equal to: `<=`

Logical Operators

- `and`: Both conditions must be True
- `or`: At least one condition must be True
- `not`: Inverts the condition

Common Mistakes

1. ❌ Using = instead of ==

```
if x = 5:  # SyntaxError
  print("x is 5")
```

2. ❌ Forgetting colons

```
if x > 5   # SyntaxError
print("x is greater than 5")
```

Practice Problems

1. Write a program that determines if a year is a leap year.
2. Create a simple grading system that gives letter grades based on numerical scores.
3. Write a program that finds the largest of three numbers.

Solutions

```
# 1. Leap year checker
def is_leap_year(year):
    if year % 4 == 0:
        if year % 100 == 0:
            if year % 400 == 0:
                return True
            return False
        return True
    return False
```

Explanation:

A year is a leap year if it is divisible by 4.

However, years divisible by 100 are not leap years, unless they are also divisible by 400.

```
# 2. Grading system
def assign_grade(score):
    if score >= 90:
        return 'A'
    elif score >= 80:
        return 'B'
    elif score >= 70:
        return 'C'
    elif score >= 60:
        return 'D'
    else:
        return 'F'

# 3. Largest of three numbers
def find_largest(a, b, c):
    if a >= b and a >= c:
        return a
```

```
    elif b >= a and b >= c:
        return b
    else:
        return c
```

10. Strings

Concept Overview

Strings are sequences of characters and are immutable in Python. They can be created using single quotes, double quotes, or triple quotes.

Key Points

1. String creation and manipulation
2. String methods
3. String indexing and slicing
4. String formatting

Examples

```
# String creation
single_quoted = 'Hello'
double_quoted = "World"
multi_line = """This is a
multi-line string"""

# String methods
text = "Hello, World!"
print(text.lower())     # hello, world!
print(text.upper())     # HELLO, WORLD!
print(text.split(','))  # ['Hello', ' World!']
print(text.strip())     # Removes leading/trailing whitespace

# Indexing and slicing
```

```
word = "Python"
print(word[0])    # P
print(word[-1])   # n
print(word[0:2])  # Py
print(word[::-1])  # nohtyP (reverses the string)

# String formatting
name = "Alice"
age = 25
# f-string (Python 3.6+)
print(f"{name} is {age} years old")
# format() method
print("{} is {} years old".format(name, age))
# % operator
print("%s is %d years old" % (name, age))
```

Common String Methods

1. `upper()`, `lower()`, `title()`
2. `strip()`, `lstrip()`, `rstrip()`
3. `replace(old, new)`
4. `split(delimiter)`
5. `join(iterable)`
6. `startswith(prefix)`, `endswith(suffix)`

Common Mistakes

1. ❌ Trying to modify strings directly

```
word = "Hello"
word[0] = 'h'  # TypeError: 'str' object does not support item assignment
```

2. ❌ Incorrect string concatenation

```
age = 25
print("Age: " + age)  # TypeError: can't concatenate str and int
```

11. Loops

Concept Overview

Loops are used to repeat a block of code multiple times. Python has two main types of loops: `for` loops and `while` loops.

Types of Loops

1. for loops - iterate over a sequence

2. while loops - repeat while a condition is true

Examples

```
# For loop with range
for i in range(5):
    print(i)  # Prints 0, 1, 2, 3, 4


# For loop with string
for char in "Python":
    print(char)  # Prints each character


# While loop
count = 0
while count < 5:
    print(count)
    count += 1


# Break and continue
for i in range(10):
    if i == 3:
        continue  # Skip 3
    if i == 8:
        break    # Stop at 8
    print(i)
```

Loop Control Statements

1. `break` - exits the loop

2. `continue` - skips to the next iteration

3. `else` - executed when loop completes normally

Common Mistakes

1. ❌ Infinite loops

```
while True:
    print("This will never end")
```

2. ❌ Off-by-one errors

```
# Wanting 1 to 10, but getting 0 to 9
for i in range(10):
    print(i)
```

Practice Problems

2. Create a multiplication table using nested loops.

3. Write a program that finds all factors of a given number.

Solutions

```
# 2. Multiplication table
def multiplication_table(n):
    for i in range(1, n+1):
        for j in range(1, n+1):
            print(f"{i*j:4}", end='')
        print()  # New line after each row


# 3. Factors finder
def find_factors(n):
    factors = []
    for i in range(1, n + 1):
        if n % i == 0:
            factors.append(i)
    return factors
```

```
# Test the functions
print("Fibonacci numbers:")
print_fibonacci()

print("\n\nMultiplication table (5x5):")
multiplication_table(5)

print("\nFactors of 24:")
print(find_factors(24))
```

12. Lists

A list is a fundamental data structure in Python that:

- Can hold multiple values at once
- Can contain any and mixed data types
- Uses indexes to access elements

Key Components:

1. Index:
   - Indicates position in the list (zero-based)
   - Uses square brackets []
   - Can be positive or negative (negative starts from end)

```
names = ["Fred", "Barney", "Wilma"]
print(names[0])   # Prints: Fred
print(names[-1])  # Prints: Wilma
```

2. Elements:
   - The actual values stored in the list
   - Can be of any data type

```
numbers = [23, 56, 44, 8]      # All integers
mixed = ["Barney", 88, 5.5]    # Mixed types
```

Common List Operations:

1. Length:

```
names = ["Fred", "Barney", "Wilma"]
print(len(names)) # Prints: 3
```

2. Looping:

```
for name in names:
  print(name)
```

3. Unpacking:

```
students = [101, "Fred", "Hamilton", 85.5]
```

```
ID, name, city, grade = students  # Assigns each value to a variable
```

13. Tuples

Tuples are similar to lists but with key differences:

- Immutable (cannot be changed after creation)
- Use parentheses () instead of square brackets
- More secure and efficient than lists

Comparison with Lists:

```
numbers = [1, 2, 3]   # List
digits = (1, 2, 3)    # Tuple

# Lists can be modified
numbers[0] = 4       # Valid

# Tuples cannot be modified
digits[0] = 4        # Raises an error
```

Single Item Tuples:

```
names = ["Fred"]    # List with single item
people = ("Fred",)  # Tuple with single item (note the comma)
words = ("Fred")    # This is a string, not a tuple!
```

14. List Subsets

Subsets are parts of a larger list, created through slicing.

Slicing Syntax:

```
numbers = [0, 1, 2, 3]

# Basic slicing
first_two = numbers[:2]   # [0, 1]
```

```
last_two = numbers[2:]    # [2, 3]


# Step slicing
every_other = numbers[0:3:2]  # [0, 2]


# Negative indexing
middle = numbers[-3:-1]   # [1, 2]


# Reversing
reversed_list = numbers[::-1] # [3, 2, 1, 0]
```

Concatenating Lists:

```
num1 = [0, 1, 2]
num2 = [3, 4, 5]
combined = num1 + num2    # [0, 1, 2, 3, 4, 5]
```

Common Pitfalls and Tips:

1. IndexError: Occurs when trying to access an index that doesn't exist

```
names = ["Fred", "Barney"]
print(names[2])    # IndexError
```

2. Tuple vs List Choice:
   - Use tuples for data that shouldn't change
   - Use lists when you need to modify the contents
3. Slicing Tips:
   - Start index is inclusive
   - End index is exclusive
   - Can use negative indices for reverse counting

When you slice a list in Python using list[start:end]:

- **Inclusive Start**: The element at the start index IS included in the result
- **Exclusive End**: The element at the end index is NOT included in the result

Visual Example

Let's look at this visually:

List: ["A", "B", "C", "D", "E"]

Index: 0 1 2 3 4

Example Slices:

1. list[1:3] gives you ["B", "C"]
   - Starts at index 1 (includes "B")
   - Stops before index 3 (excludes "D")
2. list[0:2] gives you ["A", "B"]
   - Starts at index 0 (includes "A")
   - Stops before index 2 (excludes "C")

Exercise Set 1: Basic List Operations

1. List Creation and Indexing

```
# Exercise 1.1
# Create a list called 'fruits' with the items: "apple", "banana", "cherry"
# Then answer these questions:

# Q1: How would you access "banana" using a positive index?
# Q2: How would you access "cherry" using a negative index?
# Q3: What happens if you try to access index 3? Why?

# Write your code here:
fruits = ["apple", "banana", "cherry"]

# Test your understanding:
```

```
print(fruits[1])   # Should print "banana"
print(fruits[-1])  # Should print "cherry"
# print(fruits[3])  # This will raise an IndexError
```

2. List Modification

```
# Exercise 1.2
numbers = [1, 2, 3, 4, 5]

# TODO:
# 1. Change the second number to 10
# 2. Add the number 6 to the end of the list
# 3. Remove the first number from the list

# Write your solutions here:
numbers[1] = 10
numbers.append(6)
numbers.pop(0)

print(numbers)  # What will this print?
```

Exercise Set 2: Tuple Basics

1. Tuple vs List

```
# Exercise 2.1
# Create a tuple 'coordinates' with values (3, 4)
# Create a list 'points' with values [3, 4]

# Try to modify both. What happens?
coordinates = (3, 4)
points = [3, 4]

# Uncomment these lines and try to run them:
```

```
# coordinates[0] = 5  # What happens?
# points[0] = 5      # What happens?

# Q1: Why can you change 'points' but not 'coordinates'?
# Q2: How would you create a tuple with a single value 5?
```

Exercise Set 3: List Subsets and Slicing

1. Slicing Practice

```
# Exercise 3.1
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# TODO: Use slicing to achieve these results:
# 1. Get the first three numbers
# 2. Get the last three numbers
# 3. Get every second number
# 4. Reverse the list
# 5. Get numbers from index 2 to 5

# Write your solutions here:
first_three = numbers[:3]
last_three = numbers[-3:]
every_second = numbers[::2]
reversed_list = numbers[::-1]
two_to_five = numbers[2:6]

print(f"First three: {first_three}")
print(f"Last three: {last_three}")
print(f"Every second: {every_second}")
print(f"Reversed: {reversed_list}")
print(f"Two to five: {two_to_five}")
```

Exercise Set 4: List Unpacking

1. Basic Unpacking

```
# Exercise 4.1
student = ["John Doe", 20, "Computer Science", 3.8]


# TODO: Unpack the list into variables: name, age, major, gpa
# Write your solution here:
name, age, major, gpa = student


print(f"Student: {name}, Age: {age}, Major: {major}, GPA: {gpa}")
```

Challenge Exercises

1. List Manipulation Challenge

```
# Exercise 5.1
def process_list(input_list):
    """
    1. Add the number 10 to the end of the list
    2. Change the first number to 0
    3. Reverse the list
    4. Return the second and third items as a new list
    """
    # Write your solution here
    pass


# Test your function
test_list = [1, 2, 3, 4, 5]
result = process_list(test_list)
print(f"Original list: {test_list}")
print(f"Processed result: {result}")
```

2. Tuple and List Conversion

```
# Exercise 5.2
def compare_structures():
    """
    1. Create a list of numbers 1-5
    2. Convert it to a tuple
    3. Try to sort the tuple (hint: convert back to list)
    4. Return both the original tuple and sorted tuple
    """
    # Write your solution here
    pass

# Test your function
original, sorted_tuple = compare_structures()
print(f"Original tuple: {original}")
print(f"Sorted tuple: {sorted_tuple}")
```

Answers and Explanations

(Try to solve exercises before looking!)

Exercise 1.1

```
# Q1: fruits[1]
# Q2: fruits[-1]
# Q3: IndexError because the list only has indices 0, 1, and 2
```

Exercise 1.2

```
Final list will be: [2, 10, 3, 4, 5, 6]
```

Exercise 2.1

- **Q1: Tuples are immutable, lists are mutable**
- **Q2: single_tuple = (5,) # Note the comma!**

Exercise 3.1

```
first_three = [0, 1, 2]
last_three = [7, 8, 9]
every_second = [0, 2, 4, 6, 8]
reversed_list = [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
two_to_five = [2, 3, 4, 5]
```

Challenge Exercise 5.1

```
def process_list(input_list):
  input_list.append(10)
  input_list[0] = 0
  input_list.reverse()
  return input_list[1:3]
```

Final Exam Preparation Tips

1. Practice Active Recall: Don't just read the notes. Try to solve problems without looking at solutions.

2. Create Mind Maps: Connect different concepts to see how they relate to each other.

3. Teach Others: Explaining concepts to classmates helps reinforce your understanding.

4. Time Management: During the exam, read all questions first and manage your time accordingly.

5. Common Patterns: Look for common patterns in problem-solving. Many questions can be solved using similar approaches.

6. Debug Practice: Practice finding and fixing errors in code. This is often tested in exams.

Good Luck on Your Midterm!

Remember: The key to mastering Python is practice. Try to code regularly and experiment with different concepts.