Python Fundamentals: Complete Revision Guide

1. Lists

Lists are ordered, mutable collections that can store elements of different data types.

Key Operations:

- **Creation**: Using square brackets `[]`
- **Indexing**: Accessing elements using zero-based indices
- **Slicing**: Extracting portions of lists
- **Methods**: append(), extend(), insert(), remove(), pop(), sort(), reverse()

Examples:

python

```python
# Creating lists
fruits = ["apple", "banana", "cherry"]
mixed_list = [1, "hello", 3.14, True]

# Indexing (positive and negative)
first_fruit = fruits[0]  # "apple"
last_fruit = fruits[-1]  # "cherry"

# Slicing
subset = fruits[0:2]  # ["apple", "banana"]

# Common operations
fruits.append("orange")  # Add one element
fruits.extend(["kiwi", "mango"])  # Add multiple elements
fruits.insert(1, "pear")  # Insert at specific position
fruits.remove("banana")  # Remove by value
popped_fruit = fruits.pop()  # Remove and return last element
fruits.sort()  # Sort in-place
fruits.reverse()  # Reverse in-place

# List comprehensions
squares = [x**2 for x in range(10)]  # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
even_nums = [x for x in range(20) if x % 2 == 0]  # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Practice Assignment 1:

Write a function that takes a list of numbers and returns a new list containing only the even numbers.

```python
def filter_even(numbers):
    return [num for num in numbers if num % 2 == 0]

# Test case
print(filter_even([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]))  # [2, 4, 6, 8, 10]
```

2. Functions

Functions are reusable blocks of code designed to perform specific tasks.

Key Concepts:

- **Definition**: Using `def` keyword
- **Parameters**: Input values
- **Return**: Output values
- **Scope**: Variable visibility
- **Default arguments**: Providing fallback values
- **Arbitrary arguments**: Handling variable number of arguments

Examples:

python

```python
# Basic function
def greet(name):
    return f"Hello, {name}!"

# Function with default parameter
def greet_with_default(name="Guest"):
    return f"Hello, {name}!"

# Function with multiple parameters
def add(a, b):
    return a + b

# Function with *args (variable positional arguments)
def sum_all(*numbers):
    return sum(numbers)

# Function with **kwargs (variable keyword arguments)
def create_profile(**details):
    return details

# Combining parameters
def mixed_args(a, b, *args, **kwargs):
    print(f"a: {a}, b: {b}")
    print(f"args: {args}")
    print(f"kwargs: {kwargs}")

# Examples of calling these functions
print(greet("Alice"))  # "Hello, Alice!"
print(greet_with_default())  # "Hello, Guest!"
print(greet_with_default("Bob"))  # "Hello, Bob!"
print(add(5, 3))  # 8
print(sum_all(1, 2, 3, 4, 5))  # 15
print(create_profile(name="Alice", age=25, city="New York"))  # {'name': 'Alice', 'age': 25, 'c
mixed_args(1, 2, 3, 4, 5, name="Alice", city="New York")
```

Practice Assignment 2:

Write a function that calculates the factorial of a number recursively.

python

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

# Test cases
print(factorial(5))  # 120
print(factorial(0))  # 1
```

3. Tuples

Tuples are ordered, immutable collections that can store elements of different data types.

Key Operations:

- **Creation**: Using parentheses `()`
- **Indexing**: Similar to lists
- **Unpacking**: Assigning tuple elements to variables
- **Methods**: count(), index()

Examples:

python

```python
# Creating tuples
coordinates = (10, 20)
person = ("Alice", 25, "New York")
singleton = (42,)  # Note the comma is needed for single-element tuples

# Indexing
x = coordinates[0]  # 10
name = person[0]  # "Alice"

# Tuple unpacking
name, age, city = person
x, y = coordinates

# Tuple methods
colors = ("red", "green", "blue", "red", "yellow")
red_count = colors.count("red")  # 2
green_index = colors.index("green")  # 1

# Tuples are immutable
# coordinates[0] = 15  # This will raise TypeError

# Multiple return values (actually returns a tuple)
def get_dimensions():
    return 1920, 1080

width, height = get_dimensions()
```

Practice Assignment 3:

Write a function that takes a list of tuples, each containing a student's name and score, and returns a list of names of students who scored above 70.

python

```python
def high_scorers(student_scores):
    return [name for name, score in student_scores if score > 70]

# Test case
students = [("Alice", 85), ("Bob", 65), ("Charlie", 90), ("David", 70)]
print(high_scorers(students))  # ["Alice", "Charlie"]
```

4. Lambda Functions

Lambda functions are small, anonymous functions defined using the `lambda` keyword.

Key Concepts:

- **Syntax**: `lambda arguments: expression`
- **Use Cases**: Short operations, functions as arguments

Examples:

python

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

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

# Lambda with conditionals
is_even = lambda x: True if x % 2 == 0 else False
print(is_even(4))  # True

# Lambda with lists
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))  # [1, 4, 9, 16, 25]

# Lambda with filter
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]

# Lambda with sorted
students = [("Alice", 85), ("Bob", 65), ("Charlie", 90)]
sorted_by_score = sorted(students, key=lambda x: x[1], reverse=True)
# [("Charlie", 90), ("Alice", 85), ("Bob", 65)]
```

Practice Assignment 4:

Use lambda functions with map and filter to create a list of squares of even numbers from a given list.

python

```python
def even_squares(numbers):
    return list(map(lambda x: x**2, filter(lambda x: x % 2 == 0, numbers)))

# Test case
print(even_squares([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]))  # [4, 16, 36, 64, 100]
```

5. While Loops

While loops execute a block of code as long as a condition is true.

Key Concepts:

- **Syntax**: `while condition: block`
- **Control**: break, continue, else

Examples:

python

```python
# Basic while loop
count = 0
while count < 5:
    print(count)
    count += 1
# Prints: 0, 1, 2, 3, 4

# While loop with break
count = 0
while True:
    print(count)
    count += 1
    if count >= 5:
        break
# Prints: 0, 1, 2, 3, 4

# While loop with continue
count = 0
while count < 10:
    count += 1
    if count % 2 == 0:
        continue
    print(count)
# Prints: 1, 3, 5, 7, 9

# While loop with else
count = 0
while count < 5:
    print(count)
    count += 1
else:
    print("Count reached 5")
# Prints: 0, 1, 2, 3, 4, "Count reached 5"

# Input validation with while
def get_positive_number():
    while True:
        try:
            num = float(input("Enter a positive number: "))
            if num > 0:
                return num
            print("The number must be positive.")
```

```
        except ValueError:
            print("Invalid input. Please enter a number.")
```

Practice Assignment 5:

Write a program that uses a while loop to find the first 10 Fibonacci numbers.

python

```python
def first_n_fibonacci(n):
    result = []
    a, b = 0, 1
    while len(result) < n:
        result.append(a)
        a, b = b, a + b
    return result

# Test case
print(first_n_fibonacci(10))  # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

6. For Loops

For loops iterate over a sequence (like a list, tuple, string, or range).

Key Concepts:

- **Syntax**: `for item in sequence: block`
- **Control**: break, continue, else
- **Iteration**: range(), enumerate()

Examples:

python

```python
# Basic for loop with a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
# Prints: "apple", "banana", "cherry"

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

# Range with start, stop, step
for i in range(2, 10, 2):
    print(i)
# Prints: 2, 4, 6, 8

# Loop with enumerate
fruits = ["apple", "banana", "cherry"]
for index, fruit in enumerate(fruits):
    print(f"{index}: {fruit}")
# Prints: "0: apple", "1: banana", "2: cherry"

# Loop with break
for i in range(10):
    if i == 5:
        break
    print(i)
# Prints: 0, 1, 2, 3, 4

# Loop with continue
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
# Prints: 1, 3, 5, 7, 9

# Loop with else
for i in range(5):
    print(i)
else:
    print("Loop completed")
# Prints: 0, 1, 2, 3, 4, "Loop completed"
```

```python
# Nested loops
for i in range(3):
    for j in range(2):
        print(f"({i}, {j})")
# Prints: (0,0), (0,1), (1,0), (1,1), (2,0), (2,1)
```

Practice Assignment 6:

Write a function that takes a list of numbers and returns the sum of squares of all odd numbers in the list using a for loop.

python

```python
def sum_of_odd_squares(numbers):
    total = 0
    for num in numbers:
        if num % 2 != 0:
            total += num ** 2
    return total

# Test case
print(sum_of_odd_squares([1, 2, 3, 4, 5]))  # 1^2 + 3^2 + 5^2 = 1 + 9 + 25 = 35
```

Challenging Problems

Problem 1: List Manipulation

Write a function that takes a list of integers and returns a new list where each element is the product of all numbers in the original list except the number at that index. Do this without using division.

python

```python
def products_except_self(nums):
    n = len(nums)
    result = [1] * n

    # Calculate products of all elements to the left
    left_product = 1
    for i in range(n):
        result[i] *= left_product
        left_product *= nums[i]

    # Calculate products of all elements to the right
    right_product = 1
    for i in range(n-1, -1, -1):
        result[i] *= right_product
        right_product *= nums[i]

    return result

# Test case
print(products_except_self([1, 2, 3, 4]))  # [24, 12, 8, 6]
```

Problem 2: Function Decorator

Create a decorator function that measures and prints the execution time of the decorated function.

python

```python
import time

def timer_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} executed in {end_time - start_time:.6f} seconds")
        return result
    return wrapper

@timer_decorator
def slow_function(n):
    """A deliberately slow function for testing"""
    total = 0
    for i in range(n):
        for j in range(1000000):
            total += i * j
    return total

# Test case
slow_function(5)
```

Problem 3: Tuple Processing

Write a function that takes a list of tuples, where each tuple contains a name and an arbitrary number of scores. Return a dictionary that maps each name to their average score.

python

```python
def calculate_averages(students_scores):
    result = {}
    for student in students_scores:
        name = student[0]
        scores = student[1:]
        result[name] = sum(scores) / len(scores) if scores else 0
    return result

# Test case
scores = [
    ("Alice", 85, 90, 92),
    ("Bob", 70, 65, 80, 85),
    ("Charlie", 95, 92),
    ("David",)
]
print(calculate_averages(scores))
# {'Alice': 89.0, 'Bob': 75.0, 'Charlie': 93.5, 'David': 0}
```

Problem 4: Lambda with Reduce

Use lambda functions with reduce to find the maximum element in a list of dictionaries based on a specified key.

python

```python
from functools import reduce

def find_max_by_key(items, key):
    if not items:
        return None
    return reduce(lambda a, b: a if a[key] > b[key] else b, items)

# Test case
students = [
    {"name": "Alice", "score": 85},
    {"name": "Bob", "score": 65},
    {"name": "Charlie", "score": 90},
    {"name": "David", "score": 70}
]
print(find_max_by_key(students, "score"))  # {"name": "Charlie", "score": 90}
```

Problem 5: Complex While Loop Logic

Implement a function that simulates the Collatz conjecture: if n is even, divide it by 2; if n is odd, multiply by 3 and add 1. The function should return the number of steps needed to reach 1.

python

```python
def collatz_steps(n):
    if n <= 0:
        raise ValueError("Input must be a positive integer")

    steps = 0
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        steps += 1
    return steps

# Test cases
print(collatz_steps(1))   # 0
print(collatz_steps(6))   # 8
print(collatz_steps(27))  # 111
```

Problem 6: Nested For Loop Challenge

Write a function that takes a number n and returns a list of all pairs (a, b) where a and b are positive integers, a ≤ b, and a + b ≤ n.

python

```python
def find_pairs_sum_less_equal(n):
    result = []
    for a in range(1, n):
        for b in range(a, n):
            if a + b <= n:
                result.append((a, b))
    return result

# Test case
print(find_pairs_sum_less_equal(5))
# [(1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (3, 1), (3, 2)]
```

Bonus: Combining Everything

Create a program that processes student data using all the concepts we've covered:

python

```
def process_student_data(raw_data):
    # Parse raw data into a list of student tuples
    students = []
    for line in raw_data.strip().split('\n'):
        parts = line.split(',')
        name = parts[0]
        scores = tuple(int(s) for s in parts[1:])
        students.append((name, *scores))

    # Calculate average scores using lambda and map
    averages = list(map(
        lambda student: (
            student[0],
            sum(student[1:]) / len(student[1:]) if len(student) > 1 else 0
        ),
        students
    ))

    # Find top students (scoring above 85)
    top_students = list(filter(lambda x: x[1] > 85, averages))

    # Sort students by average score
    sorted_students = sorted(averages, key=lambda x: x[1], reverse=True)

    # Group students by grade using dictionary comprehension
    def get_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'

    grades = {name: get_grade(score) for name, score in averages}

    # Count students by grade
    grade_counts = {}
    for grade in grades.values():
        if grade in grade_counts:
            grade_counts[grade] += 1
        else:
            grade_counts[grade] = 1

    return {
```

```
        'students': students,
        'averages': dict(averages),
        'top_students': dict(top_students),
        'sorted_students': sorted_students,
        'grades': grades,
        'grade_counts': grade_counts
    }

# Test with sample data
raw_data = """
Alice,92,95,88
Bob,72,65,70
Charlie,95,98,99
David,60,58,62
Eva,88,87,90
"""

result = process_student_data(raw_data)
print("Student Averages:")
for name, avg in result['sorted_students']:
    print(f"{name}: {avg:.1f} - Grade: {result['grades'][name]}")

print("\nGrade Distribution:")
for grade, count in result['grade_counts'].items():
    print(f"{grade}: {count} students")
```

Best Practices and Common Pitfalls

Lists:

- ✅ Use list comprehensions for cleaner code
- ❌ Avoid `list.append()` in a loop when you can use a list comprehension
- ❌ Don't use `[]*n` to create lists of lists (creates shallow copies)

Functions:

- ✅ Follow the Single Responsibility Principle
- ✅ Use docstrings to document your functions
- ❌ Avoid mutable default arguments (`def func(a=[]): ...`)

Tuples:

- ✅ Use tuples for immutable data or when returning multiple values

- ❌ Don't try to modify tuple elements

Lambda Functions:

- ✅ Use lambda for simple, one-line functions
- ❌ Avoid complex logic in lambda functions

While Loops:

- ✅ Always ensure the loop condition will eventually become False
- ❌ Avoid infinite loops without break conditions

For Loops:

- ✅ Use `enumerate()` when you need both index and value
- ✅ Use `zip()` to iterate over multiple sequences together
- ❌ Avoid modifying the sequence you're iterating over