Phase 01: Python Control Statements

1. Implementing Match Statements

The `match` statement is a structural pattern matching feature introduced in Python 3.10. It allows for more readable and expressive code when dealing with multiple conditions.

Syntax

```
match variable:
    case pattern1:
        # code block
    case pattern2:
        # code block
    case _:
        # default code block
```

Examples

1. **Simple Match**

    ```
    status = 404

    match status:
        case 200:
            print("OK")
        case 404:
            print("Not Found")
        case 500:
            print("Server Error")
        case _:
            print("Unknown status")
    ```

2. **Matching Tuples**

    ```
    point = (0, 0)

    match point:
        case (0, 0):
            print("Origin")
        case (x, 0):
            print(f"X-axis at {x}")
        case (0, y):
            print(f"Y-axis at {y}")
    ```

```
    case (x, y):
        print(f"Point at ({x}, {y})")
```

3. **Matching with Guards**

```
user = {"role": "admin", "name": "Alice"}

match user:
    case {"role": "admin"}:
        print("Admin user")
    case {"role": "guest"}:
        print("Guest user")
    case _ if "name" in user:
        print(f"User: {user['name']}")
    case _:
        print("Unknown user")
```

- **Readability**: Use `match` statements to improve readability over complex `if-elif-else` chains.
- **Performance**: Be aware that pattern matching can sometimes be more efficient than multiple conditional checks.
- **Use Cases**: Ideal for parsing complex data structures like JSON or dictionaries with varying formats.

2. Implementing Nested Elif Statements

Introduction

Nested `elif` statements are useful for handling multiple conditions that require hierarchical checking.

Syntax

```
if condition1:
    if condition1_1:
        # code block
    elif condition1_2:
        # code block
elif condition2:
    if condition2_1:
        # code block
    elif condition2_2:
        # code block
else:
    # default code block
```

Examples

1. **Basic Nested Elif**

```python
x = 10
y = 20

if x > 5:
    if y > 15:
        print("x is greater than 5 and y is greater than 15")
    elif y < 15:
        print("x is greater than 5 and y is less than 15")
elif x < 5:
    if y > 15:
        print("x is less than 5 and y is greater than 15")
    elif y < 15:
        print("x is less than 5 and y is less than 15")
else:
    print("x is equal to 5")
```

2. **Complex Conditions**

```python
age = 30
income = 50000

if age < 18:
    if income < 10000:
        print("Minor with low income")
    elif income >= 10000:
        print("Minor with high income")
elif 18 <= age < 65:
    if income < 10000:
        print("Adult with low income")
    elif 10000 <= income < 50000:
        print("Adult with middle income")
    else:
        print("Adult with high income")
else:
    if income < 10000:
        print("Senior with low income")
    elif 10000 <= income < 50000:
        print("Senior with middle income")
    else:
        print("Senior with high income")
```

- **Clarity**: Ensure nested `elif` statements are clear and maintainable.
- **Refactoring**: If conditions become too nested, consider refactoring into functions or using `match` statements for better readability.
- **Logical Grouping**: Group related conditions logically to avoid confusion and errors.

3. Implementing Iterators

Introduction

Iterators are objects in Python that allow traversing through all the elements of a collection or sequence.

Syntax

- **Creating an Iterator**:

  ```
  iter_obj = iter(collection)
  ```

- **Using an Iterator**:

  ```
  next(iter_obj)
  ```

Examples

1. **Using Built-in Iterator**

   ```
   my_list = [1, 2, 3, 4]
   iter_obj = iter(my_list)

   print(next(iter_obj))  # Output: 1
   print(next(iter_obj))  # Output: 2
   ```

2. **Custom Iterator Class**

   ```
   class MyRange:
       def __init__(self, start, end):
           self.current = start
           self.end = end

       def __iter__(self):
           return self

       def __next__(self):
           if self.current >= self.end:
               raise StopIteration
           current = self.current
           self.current += 1
           return current

   my_range = MyRange(1, 5)
   for number in my_range:
       print(number)  # Output: 1 2 3 4
   ```

- **Efficiency**: Iterators can improve memory efficiency by generating elements on-the-fly rather than storing entire collections in memory.
- **Custom Iterators**: Use custom iterators to encapsulate complex iteration logic.
- **Generators**: Consider using generators (`yield` keyword) for simpler and more readable code.

4. Applying Break and Continue Statements

The `break` and `continue` statements are used to control the flow of loops. The `break` statement exits the loop prematurely, while the `continue` statement skips the rest of the code inside the loop for the current iteration and moves to the next iteration.

Syntax

- **Break**:

  ```
  for item in collection:
      if condition:
          break
      # code block
  ```

- **Continue**:

  ```
  for item in collection:
      if condition:
          continue
      # code block
  ```

Examples

1. **Using Break in a Loop**

   ```
   for i in range(10):
       if i == 5:
           break
       print(i)
   # Output: 0 1 2 3 4
   ```

2. **Using Continue in a Loop**

   ```
   for i in range(10):
       if i % 2 == 0:
           continue
       print(i)
   # Output: 1 3 5 7 9
   ```

3. **Nested Loops with Break**

```
for i in range(3):
    for j in range(3):
        if j == 1:
            break
        print(f"i: {i}, j: {j}")
# Output:
# i: 0, j: 0
# i: 1, j: 0
# i: 2, j: 0
```

4. **Nested Loops with Continue**

```
for i in range(3):
    for j in range(3):
        if j == 1:
            continue
        print(f"i: {i}, j: {j}")
# Output:
# i: 0, j: 0
# i: 0, j: 2
# i: 1, j: 0
# i: 1, j: 2
# i: 2, j: 0
# i: 2, j: 2
```

- **Break Usage**: Use `break` to exit loops when a condition is met, preventing unnecessary iterations.
- **Continue Usage**: Use `continue` to skip the current iteration and proceed to the next one when certain conditions are met.
- **Nested Loops**: When using `break` and `continue` in nested loops, be clear about which loop they are intended to control.

5. Implementing If-Else Statements

The `if-else` statement is used to execute code based on whether a condition is true or false.

Syntax

```
if condition:
    # code block for true condition
else:
    # code block for false condition
```

Examples

1. **Simple If-Else**

```python
age = 18

if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

2. **If-Elif-Else**

```python
score = 85

if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
else:
    print("Grade: F")
```

3. **Nested If-Else**

```python
num = 10

if num > 0:
    if num % 2 == 0:
        print("Positive even number")
    else:
        print("Positive odd number")
else:
    if num % 2 == 0:
        print("Negative even number")
    else:
        print("Negative odd number")
```

- **Readability**: Ensure that `if-else` statements are clear and readable. Avoid deep nesting.
- **Boolean Logic**: Simplify conditions using boolean logic to make code more concise.
- **Ternary Operator**: For simple conditional assignments, consider using the ternary operator for brevity:

  ```python
  result = "Adult" if age >= 18 else "Minor"
  ```

---

6. Implementing While/Do-While Loops

The `while` loop repeatedly executes a block of code as long as the condition is true. Python does not have a built-in `do-while` loop, but similar behavior can be mimicked.

Syntax

- **While Loop**:

  ```
  while condition:
      # code block
  ```

- **Mimicking Do-While Loop**:

  ```
  while True:
      # code block
      if not condition:
          break
  ```

Examples

1. **Simple While Loop**

   ```
   count = 0

   while count < 5:
       print(count)
       count += 1
   # Output: 0 1 2 3 4
   ```

2. **Do-While Loop Behavior**

   ```
   count = 0

   while True:
       print(count)
       count += 1
       if count >= 5:
           break
   # Output: 0 1 2 3 4
   ```

3. **Using While with Else**

   ```
   count = 0

   while count < 5:
       print(count)
   ```

```
    count += 1
else:
    print("Loop ended")
# Output: 0 1 2 3 4
#         Loop ended
```

- **Condition Check**: Ensure that the loop condition will eventually become false to avoid infinite loops.
- **Break in Do-While**: When mimicking `do-while`, ensure that the `break` condition is well-defined to prevent infinite execution.
- **Use Cases**: Use `while` loops for indefinite iteration when the number of iterations is not known beforehand.

7. Implementing For Loops

The `for` loop in Python is used for iterating over a sequence (such as a list, tuple, or string).

Syntax

```
for item in sequence:
    # code block
```

Examples

1. **Iterating Over a List**

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

    for fruit in fruits:
        print(fruit)
    # Output: apple banana cherry
    ```

2. **Iterating Over a String**

    ```
    word = "hello"

    for letter in word:
        print(letter)
    # Output: h e l l o
    ```

3. **Using Range in For Loop**

    ```
    for i in range(5):
        print(i)
    ```

```
# Output: 0 1 2 3 4
```

4. **Nested For Loops**

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

- **Sequences**: Use `for` loops for iterating over sequences where the number of iterations is known.
- **Efficiency**: Be mindful of the efficiency, especially in nested loops.
- **Itertools**: For advanced iteration patterns, consider using the `itertools` module for better performance and readability.

8. Implementing the Range Function in Loops

The `range` function generates a sequence of numbers, which is commonly used in loops for a specified number of iterations.

Syntax

```
range(stop)
range(start, stop)
range(start, stop, step)
```

Examples

1. **Basic Range Usage**

```
for i in range(5):
    print(i)
# Output: 0 1 2 3 4
```

2. **Range with Start and Stop**

```
for i in range(2, 6):
    print(i)
# Output: 2 3 4 5
```

3. **Range with Step**

   ```
   for i in range(1, 10, 2):
       print(i)
   # Output: 1 3 5 7 9
   ```

4. **Negative Step**

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

- **Memory Efficiency**: `range` is memory efficient as it generates numbers on the fly.
- **Start and Stop**: Specify `start` and `stop` for customized ranges.
- **Step**: Use the `step` parameter to skip values or reverse the range.