Lecture 03 - Control Structures

Overview

Control structures are essential in programming: they allow to control the flow of the code.

In Python, the primary control structures are `if`, `while`, and `for` loops. These structures are crucial in decision-making processes and repetitive tasks.

This notebook covers:

- `if` statements
- `while` loops for repeated execution based on a condition
- `for` loops for iterating over sequences
    - **Counter-based loops**
    - **List comprehension**

1. The `if` Statement

The `if` statement executes a block of code only if a certain condition is `True`.

Such **conditional statements** consider the specific value of a variable at the time of execution and determine the outcome based on a logical operation.

Structure

```
If CONDITON HOLDS:
        OUTCOME 1
    Elif OTHER CONDITION HOLDS:
        OUTCOME 2
    Else:
        OUTCOME 3
```

Note: check the `tabs`

In [ ]:

```python
# Example: withdrawal
balance = 500
withdrawal_amount = 800

if balance >= withdrawal_amount:
    balance -= withdrawal_amount
    print(f"Withdrawal successful! New balance: {balance}")
else:
    print("Insufficient funds.")
```

In [ ]:

```
# Input values
savings = float(input("Enter your savings amount in dollars: "))
monthly_income = float(input("Enter your monthly income in dollars: "))
debt = float(input("Enter your debt amount in dollars: "))

# Applying the conditions using if, elif, and else statements
if savings > 50000 and debt == 0:
    print("You should invest in stocks.")
elif debt > 0 and savings > 20000 and monthly_income > 4000:
    print("You should pay off debt aggressively.")
elif savings < 20000 and debt > 10000:
    print("You should save more and minimize spending.")
else:
    print("You should create a budget and build an emergency fund.")
```

2. The `while` Loop

The `while` loop repeats a block of code as long as a specified condition is true.

2.1 Structure

```
While CONDITION HOLDS:
        ACTION(s)
```

In [ ]:

```python
# Example: Simulating a simple ATM withdrawal process
balance = 1000
withdrawal_attempts = 0
withdrawal_amount = 200

while balance > 0 and withdrawal_attempts < 3:
    if balance >= withdrawal_amount:
        balance -= withdrawal_amount
        print(f"Withdrawal successful! New balance: {balance}")
    else:
        print("Insufficient funds.")
    withdrawal_attempts += 1
```

2.2 When using a `while` loop:

1. **Loop Condition and Termination**: Ensure the loop has a valid exit condition that will eventually become `False`.

In [ ]:
```
count = 0
while count < 5:  # Loop will terminate when count reaches 5
    print(f"Count is {count}")
    count += 1
```

If the loop condition (e.g., `count < 5`) is never met, the loop will not stop naturally.*

2. **Update Loop Variable**: Ensure the variable controlling the loop is updated to prevent infinite loops.|

In [ ]:
```
balance = 1000
monthly_payment = 200

while balance > 0:
    balance -= monthly_payment  # Update balance each iteration
    print(f"Remaining balance: ${balance}")
```

> *The balance is reduced each time until it reaches zero, terminating the loop.*

3. **Break and Exit Conditions**: Use `break` to exit early when certain conditions are met.

In [ ]:
```
while True:
    user_input = input("Type 'exit' to stop: ")
    if user_input == 'exit':
        print("Exiting the loop.")
        break
```

> *The loop terminates immediately when the user types `'exit'`.*

4. **Efficiency and Performance**: Avoid heavy computations inside the loop to maintain performance.

In [ ]:
```
n = 1000000
i = 0
while i < n:
    i += 1  # Efficient loop, only counting
# Avoid putting expensive operations here
print(f"Loop completed {n} iterations.")
```

> *The loop is efficient and avoids unnecessary complex calculations inside.*

5. **Edge Case Handling**: Plan for edge cases by adding safety conditions, such as a maximum number of iterations.

In [ ]:

```
loan_balance = 5000
monthly_payment = 300
max_months = 60  # Safety condition to prevent infinite loop
months = 0

while loan_balance > 0 and months < max_months:
    loan_balance -= monthly_payment
    months += 1
    if loan_balance < 0:
        loan_balance = 0
        break

print(f"Loan repaid in {months} months.")
```

> *The loop stops if either the loan is repaid or the maximum number of months is reached.*

2.3 Full example: Loan Repayment Simulation

Set-up: A user takes out a loan of $\$100,000$ with an annual interest rate of 5%. The user makes fixed monthly payments of $\$1,500$.

Goal:

1. Calculate how many months it will take to pay off the loan.
2. Track how much total interest is paid by the time the loan is fully repaid.

To consider: In this example, we simulate the process of repaying a loan with monthly payments.

The goal is to calculate how long it will take to fully repay the loan, considering:

- A principal loan amount.
- A fixed monthly payment.
- A monthly interest rate (compound interest).

We use a `while` loop to simulate the monthly loan repayment process until the loan is fully repaid. The loop keeps running as long as the loan balance is greater than zero.

In [ ]:

```
# Loan parameters
principal = 100000  # Initial loan amount in dollars
annual_interest_rate = 0.05  # 5% annual interest
monthly_payment = 1500  # Monthly payment in dollars
```

In [ ]:

```
# Calculated monthly interest rate
monthly_interest_rate = annual_interest_rate / 12
```

In [ ]:

```
# Initialize variables
loan_balance = principal  # Remaining loan balance starts as the princi
total_interest_paid = 0  # Track the total interest paid
months = 0  # Track the number of months needed to repay the loan
```

In [ ]:

```
# Loop until the loan is repaid
while loan_balance > 0:
    # Calculate interest for the current month
    monthly_interest = loan_balance * monthly_interest_rate
    total_interest_paid += monthly_interest

    # Update loan balance by subtracting the monthly payment (minus the
    loan_balance = loan_balance + monthly_interest - monthly_payment

    # Increment the number of months
    months += 1

    # If the loan balance becomes less than the monthly payment in the
    # pay off the remaining balance and break the loop
    if loan_balance < monthly_payment:
        total_interest_paid += loan_balance * monthly_interest_rate  #
        loan_balance = 0  # Set the balance to 0
        months += 1  # Add the final month
```

In [ ]:

```
# Output the result
print(f"It will take {months} months to repay the loan.")
print(f"Total interest paid over the life of the loan: ${total_interest
```

3. The `for` Loop

The `for` loop is used to iterate over a sequence (like a `list`, `tuple`, `string`, etc.) and execute a block of code for each item in the sequence.

3.1 Structure

```
For ITEM in ITERABLE:
        ACTION(s)
```

3.1 Structure

```
For ITEM in ITERABLE:
        ACTION(s)
```

- `item`: This is a variable that takes the value of each element in the sequence on every iteration.
- `iterable`: This is any Python object that can return one item at a time, like `lists`, `tuples`, `strings`, `ranges`, etc.

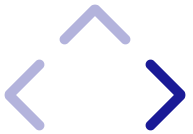
3.1 Structure

```
For ITEM in ITERABLE:
        ACTION(s)
```

- `item`: This is a variable that takes the value of each element in the sequence on every iteration.
- `iterable`: This is any Python object that can return one item at a time, like `lists`, `tuples`, `strings`, `ranges`, etc.

How It Works

- The `for` loop goes through each item in the iterable one by one.
- On each iteration, the item variable is assigned the next value in the sequence, and the block of code inside the loop is executed.
- This continues until all items in the iterable have been processed.



In [ ]:

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

for fruit in fruits:
    print(fruit)
```

3.2 Looping over `lists`, `dictionnaries` and `strings`

In [ ]:
```
l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for element in l[2:5]:
    print (element ** 2)
```

In [ ]:
```
d = {
    "program": "Master of Business Administration",
    "year": 2025,
    "number_of_students": 42,
    "average_age": 22,
    "specializations": ["Finance", "Marketing", "Data Science", "Strate
}
for item in d.items():
    print (item)
```

In [ ]:
```
for value in d.values():
    print (type(value))
```

In [ ]:

```python
# Example: Calculating the total balance from a list of transactions
transactions = [100, -50, 200, -75, 150]
total_balance = 0

for transaction in transactions:
    total_balance += transaction

print("Total balance after all transactions:", total_balance)
```

In [ ]:

```
# Looping over strings
message = "Hello"

for char in message:
    print(char)
```

3.3 Counter-based looping

The `range()` function is often used in `for` loops when you need to loop a specific number of times or generate a sequence of numbers.

In [ ]:
```
r = range(0, 8, 1)
r
```

In [ ]:
```
list(r)
```

In [ ]:
```
type (r)
```

In [ ]:

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

- `range(5)` generates a sequence of numbers from 0 to 4.
- On each iteration, the variable i takes the value of the next number in the range.

In [ ]:

```
for i in range(2,5):
    print (l[i] ** 2)
```

3.4 Nested `for` loops

`for` loops can be nested to iterate over multi-dimensional data structures like lists of lists or matrices:

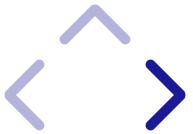
In [ ]:

```python
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for row in matrix:
    for item in row:
        print(item, end=" ")
    print()  # Newline after each row
```

In [ ]:
```python
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

for row in matrix:
    for item in row:
        print(item, end=" ")
    print()  # Newline after each row
```

In this case:

- The first line: `for row in [[1, 2], [3, 4], [5, 6]]` iterates over each row of the matrix.
- The second line: `for item in row` iterates over each item in the row.
- The `item` expression adds the individual elements to the new list.



3.5 `break` and `continue` and `else`

- `break`: Exits the loop prematurely when a certain condition is met.
- `continue`: Skips the current iteration and moves on to the next iteration.
- `else`: Executes after the loop completes normally (i.e., when the loop is not terminated by a break statement).

In [ ]:
```python
for num in range(1, 10):
    if num == 5:
        break  # Exit the loop when num equals 5
    print(num)
```

In [ ]:
```python
for num in range(1, 6):
    if num == 3:
        continue  # Skip the iteration when num equals 3
    print(num)
```

In [ ]:
```
for num in range(1, 5):
    print(num)
else:
    print("Loop completed.")
```

In [ ]:
```
# If the loop is interrupted by a break, the else block is not executed
for num in range(1, 5):
    if num == 3:
        break
    print(num)
else:
    print("Loop completed.")
```

3.6 Looping and conditioning

In [ ]:

```
for i in range (1,10):
    if i % 2 == 0:
        print (i, 'is even')
    elif i % 3 == 0:
        print (i, 'is a multiple of 3')
    else:
        print (i, 'is odd')
```

3.7 List comprehension

List comprehension is a concise way to create lists in Python. It combines a `for` loop and optional conditions into a single line of code, allowing to generate `lists` quickly and efficiently.

It is not only shorter but often more readable and **efficient computationally** than traditional `for` loops when dealing with list generation.

Basic Syntax of List Comprehension

```
new_list = [expression for item in iterable if condition]
```

- **`expression`**: The value to append to the new list.
- **`item`**: The variable representing each element in the iterable (e.g., list, string, or range).
- **`iterable`**: Any sequence or collection being iterated over (e.g., a list, tuple, string, or range).
- **`condition`**: (Optional) A filter that includes only certain elements from the iterable. The condition is an `if` statement.

In [ ]:

```python
#  Traditional `for` Loop:
numbers = [1, 2, 3, 4, 5]
squares = []

for number in numbers:
    squares.append(number ** 2)

print(squares)
```

In [ ]:

```python
# Equivalent List Comprehension:
squares = [number ** 2 for number in [1, 2, 3, 4, 5]]
print(squares)
```

In this case:

- The `for number in [1, 2, 3, 4, 5]` part is the iteration.
- The `number ** 2` part is the expression that generates the square of each number.

List comprehesions with condition

`if` statement can be included in a list comprehension to filter elements.

In [ ]:

```python
# Traditional `for` Loop with Condition:
numbers = [1, 2, 3, 4, 5, 6]
evens = []

for number in numbers:
    if number % 2 == 0:
        evens.append(number)

print(evens)
```

In [ ]:

```
# Equivalent list comprehension with condition:
evens = [number for number in [1, 2, 3, 4, 5, 6] if number % 2 == 0]
print(evens)
```

In [ ]:

```
# List comprehension with multiple conditions
filtered_numbers = [number for number in range(1, 21) if number % 2 ==
print(filtered_numbers)
```

List comprehension with function calls

Function calls can be used in the expression part of a list comprehension.

In [ ]:
```
def square(number):
    return number ** 2
```

In [ ]:
```
numbers = [1, 2, 3, 4, 5]
squares = [square(number) for number in numbers]
```

Dictionary with list comprehension

List comprehensions can be used to create dictionaries using the `dict()` constructor.

In [ ]:

```
names = ['Alice', 'Bob', 'Charlie']
name_length_dict = {name: len(name) for name in names}

print(name_length_dict)
```

When to use list comprehensions

- **Conciseness**: Generate lists quickly and concisely.
- **Readability**: For simple iterations, list comprehension is often more readable than multiple lines of `for` loops.
- **Efficiency**: List comprehensions are more efficient than traditional loops due to optimizations in Python's internal implementation.

When Not to Use List Comprehension

- **Complex Logic**: If the logic inside the list comprehension becomes too complex (e.g., multiple `if` conditions, nested loops), it may become less readable. In such cases, traditional loops may be clearer.
- **Side Effects**: List comprehensions should not be used if the logic involves side effects (e.g., modifying external variables or data structures), as they are primarily designed for list generation, not process flow control.