Unit 2
Decision and Loops

Er. Narayan Sapkota
M.Sc. Computational Science and Engineering

Kathmandu University (Visiting Faculty)

December 11, 2024

Table of Contents

The Control Flow Statements (1)

The control flow statements in Python Programming Language are

- Sequential Control Flow: In this flow, statements are executed one after the other, in the order they appear in the program.
- Decision Control Flow: Based on a condition (True or False), the program decides which block of code to execute. Examples include `if`, `if...else`, and `if...elif...else`.
- Loop Control Flow: This structure allows repeating a block of code multiple times until a specified condition is met. Examples are `for` and `while` loops. These are also referred to as repetition or iteration statements.

The Control Flow Statements (2)

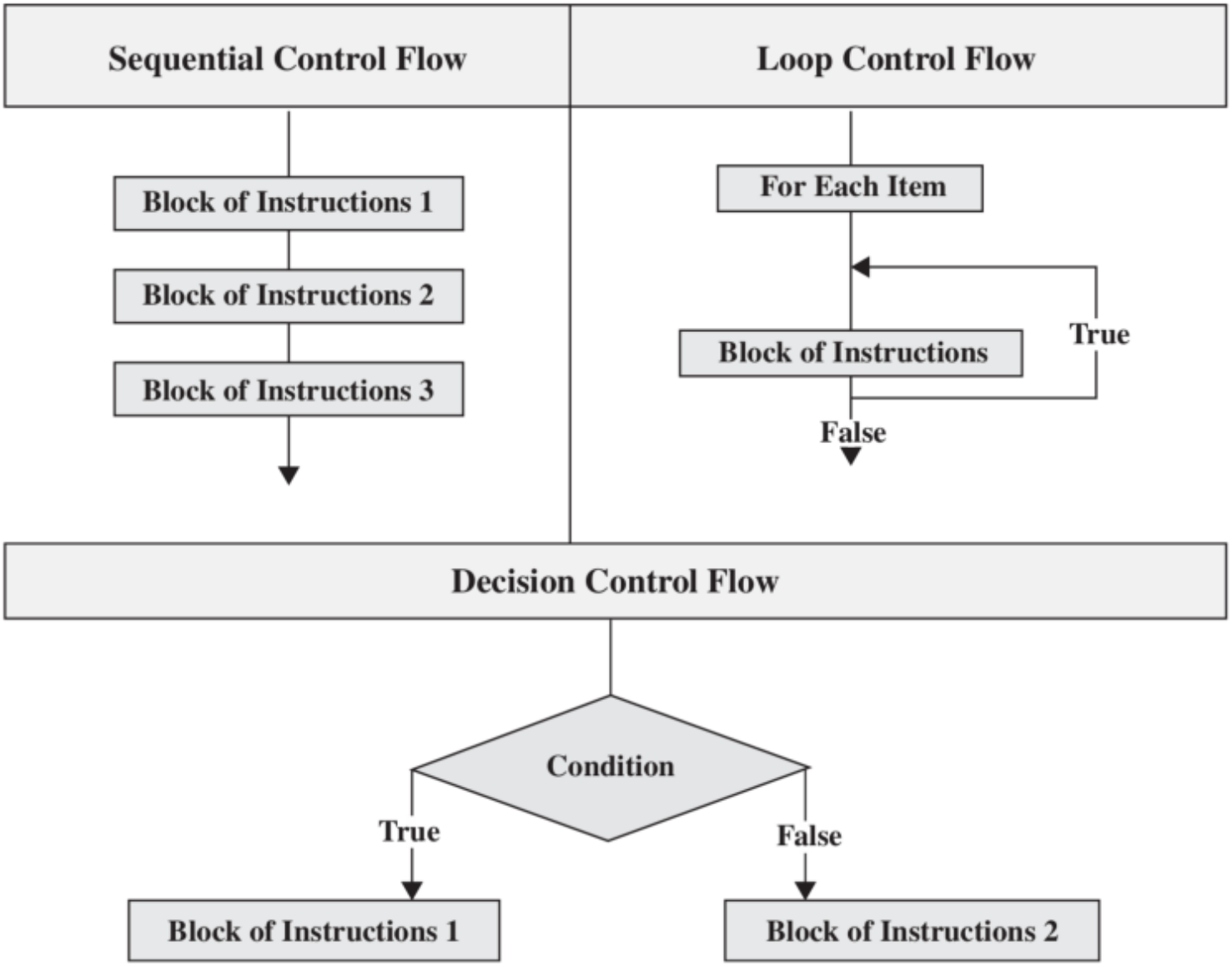
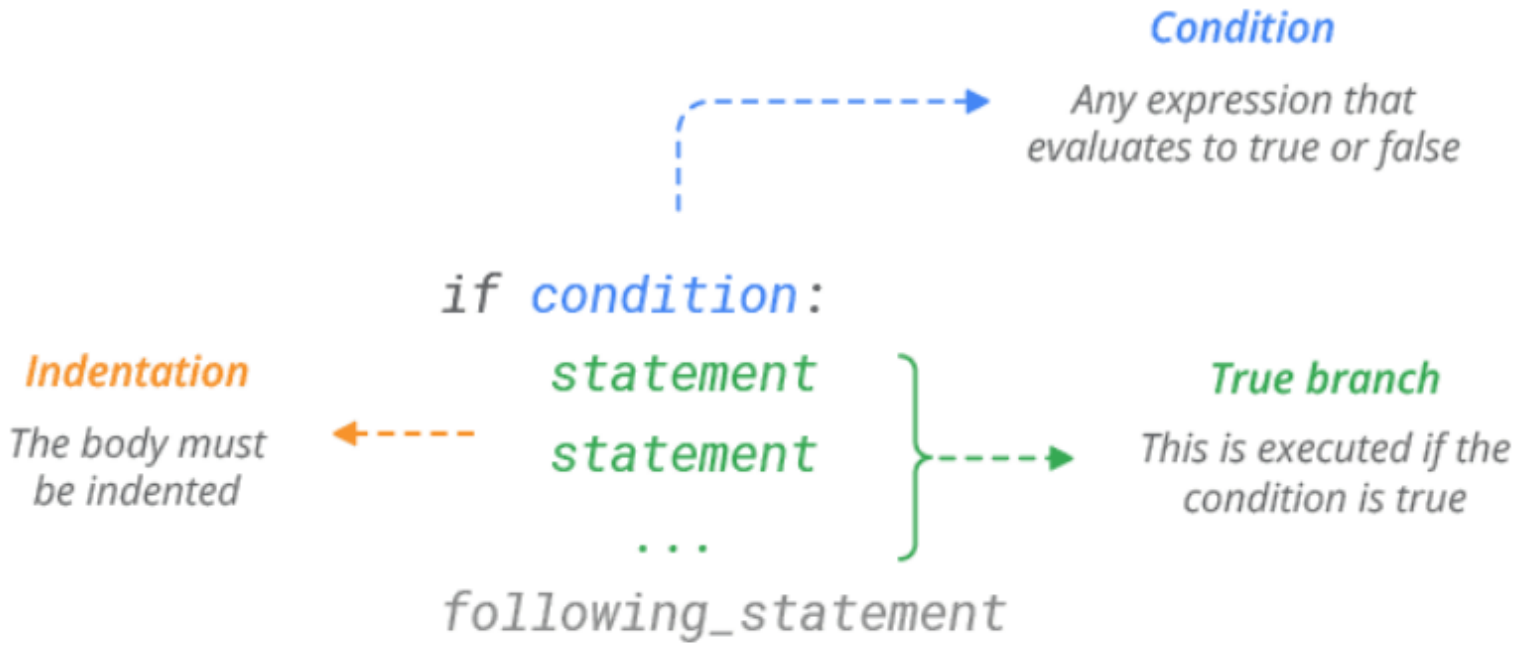


Table of Contents

The `if` Statement (1)

- The `if` statement starts with the `if` keyword and ends with a colon (:). The condition in the `if` statement must be a Boolean expression.
- If the condition evaluates to `True`, the code block indented under the `if` statement is executed.
- If the condition evaluates to `False`, the block is skipped.
- Indentation defines the scope of the `if` block, and the first non-indented statement marks the end of the block.
- You do not need to use the == operator explicitly for a variable to check if it evaluates to `True`. The variable itself can serve as the condition.

The if Statement (2)



Program 1: Program to Check if a Number is Positive

```
number = int(input("Enter a number"))
if number >= 0:
  print("The number entered is positive")
```

The `if` Statement (3)

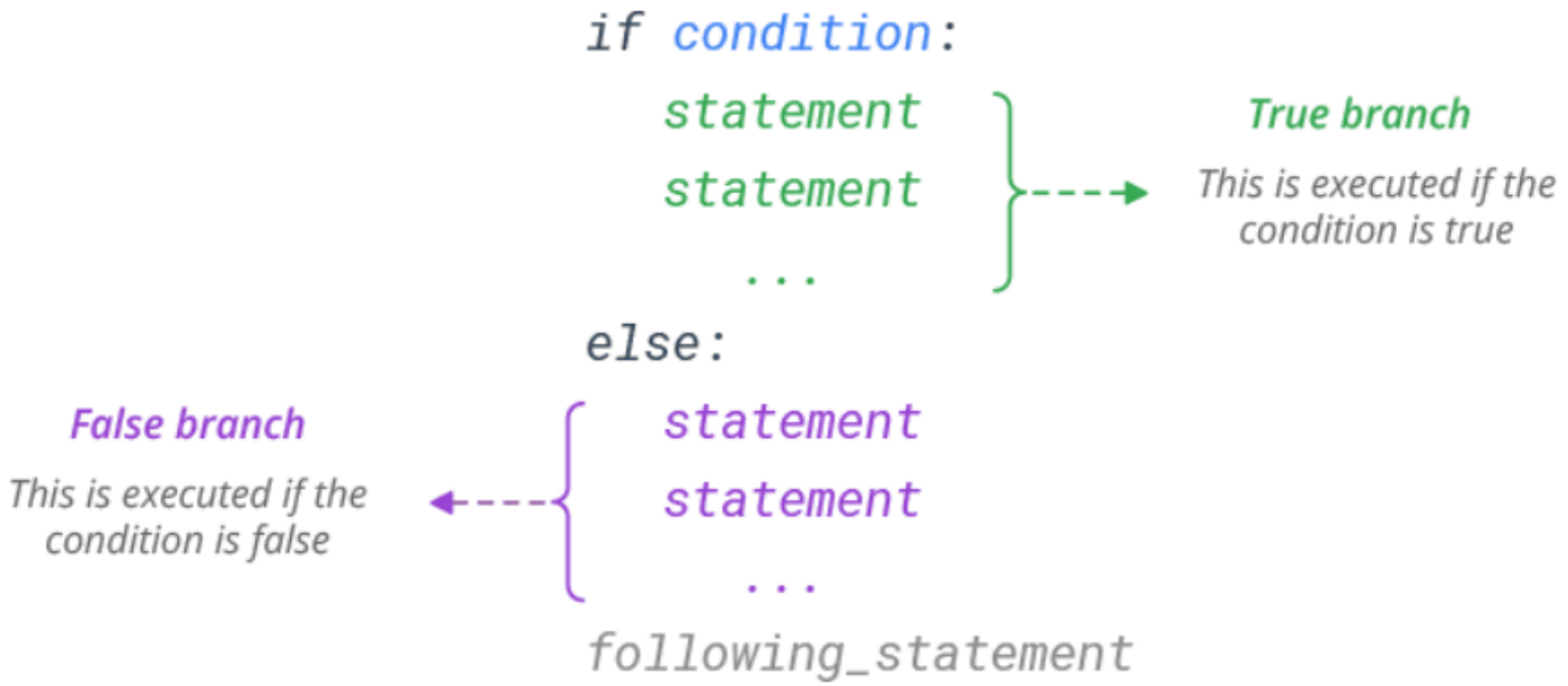
Output:

```
Enter a number:  67
The number entered is positive
```

The `if...else` Statement (1)

- An `if` statement can be followed by an optional `else`. The `else` block has no condition.
- If the `Boolean_Expression` is `True`, the `if` block executes; otherwise, the `else` block executes.
- The `if...else` statement provides a two-way decision. Both `if` and `else` must align at the same column.
- Indentation separates the blocks, and control moves to the next statement after execution.

The `if...else` Statement (2)



The if...else Statement (3)

Program 2: Check if a Number is Odd or Even

```
number = int(input("Enter a number"))
if number % 2 == 0:
  print(f"{number} is Even number")
else:
  print(f"{number} is Odd number")
```

Output:

```
Enter a number:  45
45 is Odd number
```

Program 3: Greater of Two Numbers

```
number_1 = int(input("Enter the first number"))
number_2 = int(input("Enter the second number"))
if number_1 > number_2:
  print(f"{number_1} is greater than {number_2}")
else:
  print(f"{number_2} is greater than {number_1}")
```

The if...else Statement (4)

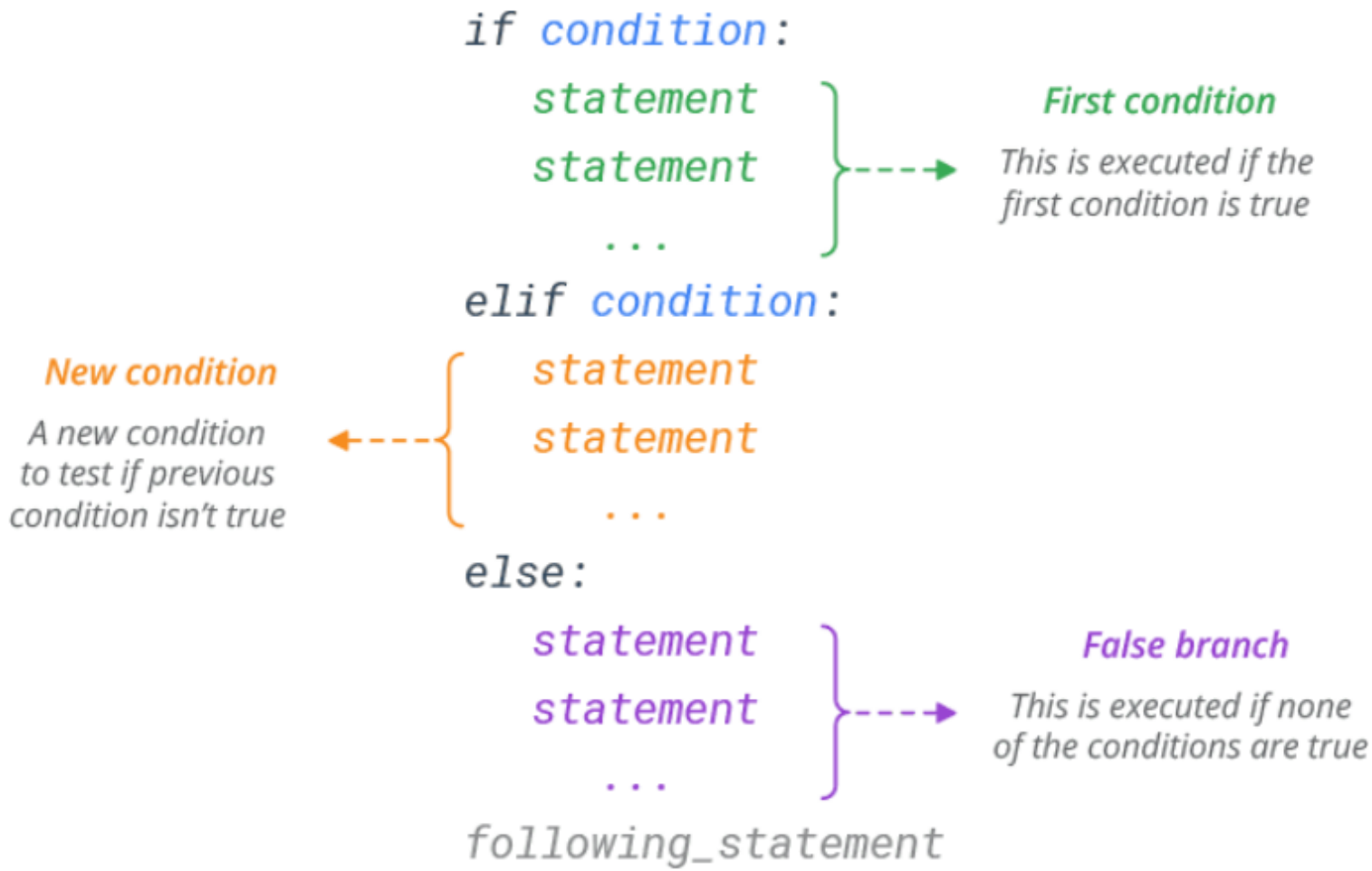
Output:

```
Enter the first number:  8
Enter the second number:  10
10 is greater than 8
```

The `if...elif...else` Statement (1)

- The `if...elif...else` statement allows multiple conditions to be checked.
- `if` tests the first condition. If the `if` condition is `False`, `elif` (else if) checks the next condition.
- If none of the `if` or `elif` conditions are `True`, the `else` block executes.
- Only one block executes: the first `True` condition or the `else` block.
- Indentation is used to define the blocks, and `if`, `elif`, and `else` must align.

The if...elif...else Statement (2)



The if...elif...else Statement (3)

Program 4: Positive or Negative number

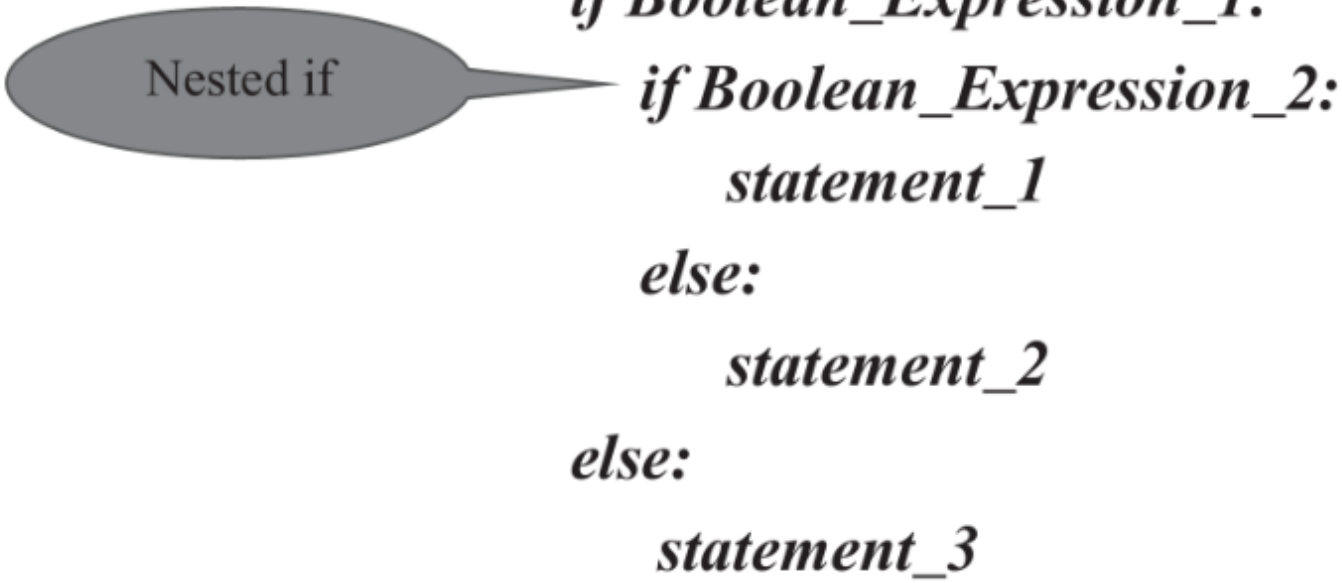
```
number = int(input("Enter a number"))
if number > 0:
  print(f"{number} is positive")
elif number < 0:
  print(f"{number} is negative")
else:
  print(f"{number} is zero")
```

Output:

```
Enter a number:  5
5 is positive
```

Nested `if` statement (1)

- A nested `if` statement is an `if` statement inside another `if` statement.
- It allows you to check multiple conditions at different levels.
- If the outer `if` condition is `True`, the inner `if` statement is evaluated.
- If the outer condition is `False`, the inner `if` is skipped.



Nested if statement (2)

Program 5: Check if a Number is Positive and Even or Odd

```
number = int(input("Enter a number"))
if number > 0:   # Outer if
  if number % 2 == 0:   # Nested if
    print(f"{number} is a positive even number")
  else:
    print(f"{number} is a positive odd number")
else:
  print(f"{number} is not positive")
```

Output:

```
Enter a number:  8
8 is a positive even number
```

Nested `if` statement (3)

Write a Python program to prompt for a score between 0.0 and 1.0. If the score is out of range, print an error. If the score is between 0.0 and 1.0, print a grade using the following table:

Score	Grade
≥ 0.9	A
≥ 0.8	B
≥ 0.7	C
≥ 0.6	D
< 0.6	F

Nested if statement (4)

```
score = float(input("Enter your score: "))
if score < 0 or score > 1:
  print('Wrong Input')
elif score >= 0.9:
  print('Your Grade is "A"')
elif score >= 0.8:
  print('Your Grade is "B"')
elif score >= 0.7:
  print('Your Grade is "C"')
elif score >= 0.6:
  print('Your Grade is "D"')
else:
  print('Your Grade is "F"')
```

Output:

```
Enter your score:  0.92
Your Grade is "A"
```

Nested if statement (5)

Program 6: Check if a Year is a Leap Year

```
year = int(input('Enter a year: '))

if year % 4 == 0:
  if year % 100 == 0:
    if year % 400 == 0:
      print(f'{year} is a Leap Year')
    else:
      print(f'{year} is not a Leap Year')
  else:
    print(f'{year} is a Leap Year')
else:
  print(f'{year} is not a Leap Year')
```

Output:

```
Enter a year:  2014
2014 is not a Leap Year
```

The `switch` Statement (1)

- Python does not have a built-in `switch` statement like some other programming languages (C, Java, etc.).
- Similar functionality can be achieved using `if...elif...else` blocks or a dictionary.
- The `if...elif...else` structure can simulate a `switch` by checking multiple conditions one by one.
- Alternatively, a dictionary can map specific values to corresponding actions or functions, providing a more efficient solution.

The switch Statement (2)

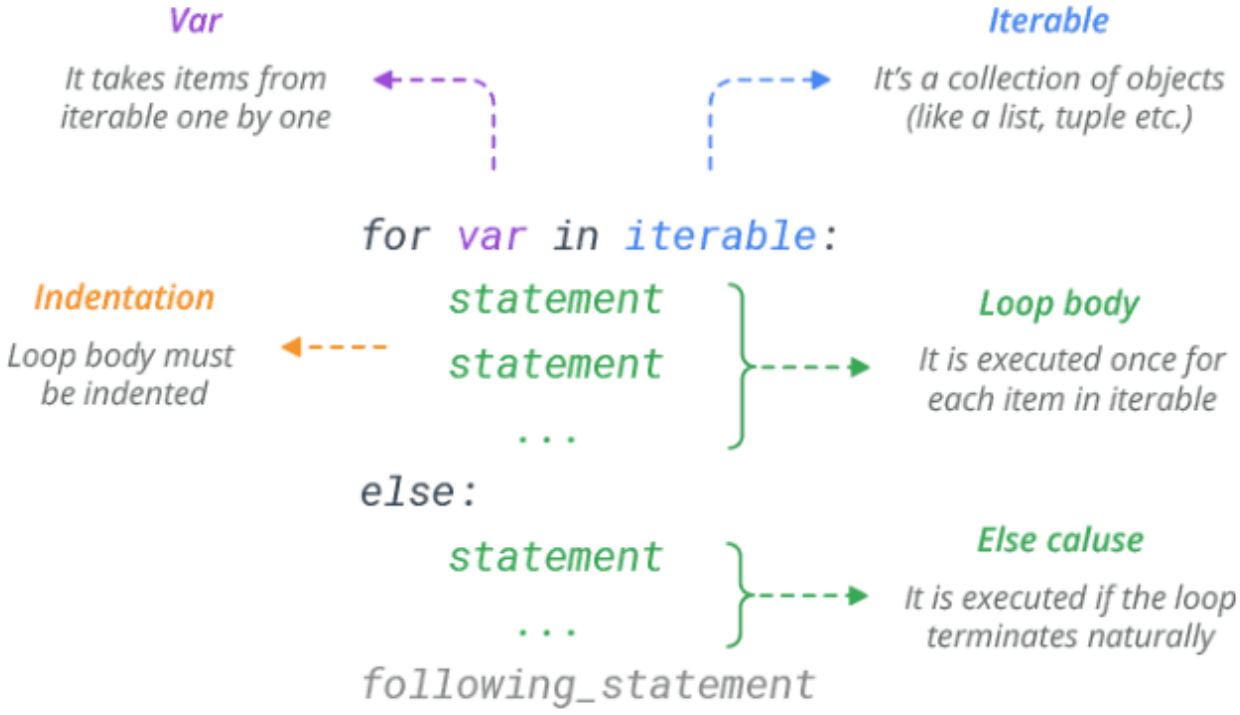
```
def switch_case(value):
  if value == 'a':
    print("You selected 'a'")
  elif value == 'b':
    print("You selected 'b'")
  elif value == 'c':
    print("You selected 'c'")
  else:
    print("Invalid selection")

switch_case('b')  # Output: You selected 'b'
```

Table of Contents

for loop (1)

- The `for` loop in Python is used to iterate over a sequence (like a list, tuple, string, or range). It executes a block of code repeatedly for each item in the sequence.
- Python's `for` loop simplifies iteration as it directly supports iteration over collections.



for loop (2)

Program 7: Iterating Over a List

```
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
  print(fruit)
```

Output:

```
apple
banana
cherry
```

- Using range in a for loop:

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

  - start (optional): This value indicates the beginning of the sequence. If not specified, the sequence starts from 0 by default.
  - stop: This value defines the end of the sequence, but the sequence will not include this number itself.

for loop (3)

- step (optional): This value indicates the difference between every two consecutive numbers in the sequence. It can be both positive and negative, but not zero.

Program 8: Using range in a for loop

```
for i in range(3):
  print(i)
```

Output:

```
0
1
2
```

for loop (4)

Write a Program to Find the Sum of All Odd and Even Numbers up to a Number Specified by the User.

```
n = int(input("Enter a number: "))
sum_even = 0
sum_odd = 0

# Loop through all numbers from 1 to n
for i in range(1, n + 1):
  if i % 2 == 0:
    sum_even += i
  else:
    sum_odd += i

print(f"The sum of even numbers up to {n} is: {sum_even}")
print(f"The sum of odd numbers up to {n} is: {sum_odd}")
```

for loop (5)

Output:

```
Enter a number:  5
The sum of even numbers up to 5 is:  6
The sum of odd numbers up to 5 is:  9
```

for loop (6)

Write a Program to Find the Factorial of a Number using for loop.

```
num = int(input("Enter a number: "))

# Initialize the factorial to 1
factorial = 1

if num < 0:
  print("Factorial is not defined for negative numbers.")
elif num == 0:
  print("The factorial of 0 is 1.")
else:
  for i in range(1, num + 1):
    factorial *= i
  print(f"The factorial of {num} is {factorial}.")
```

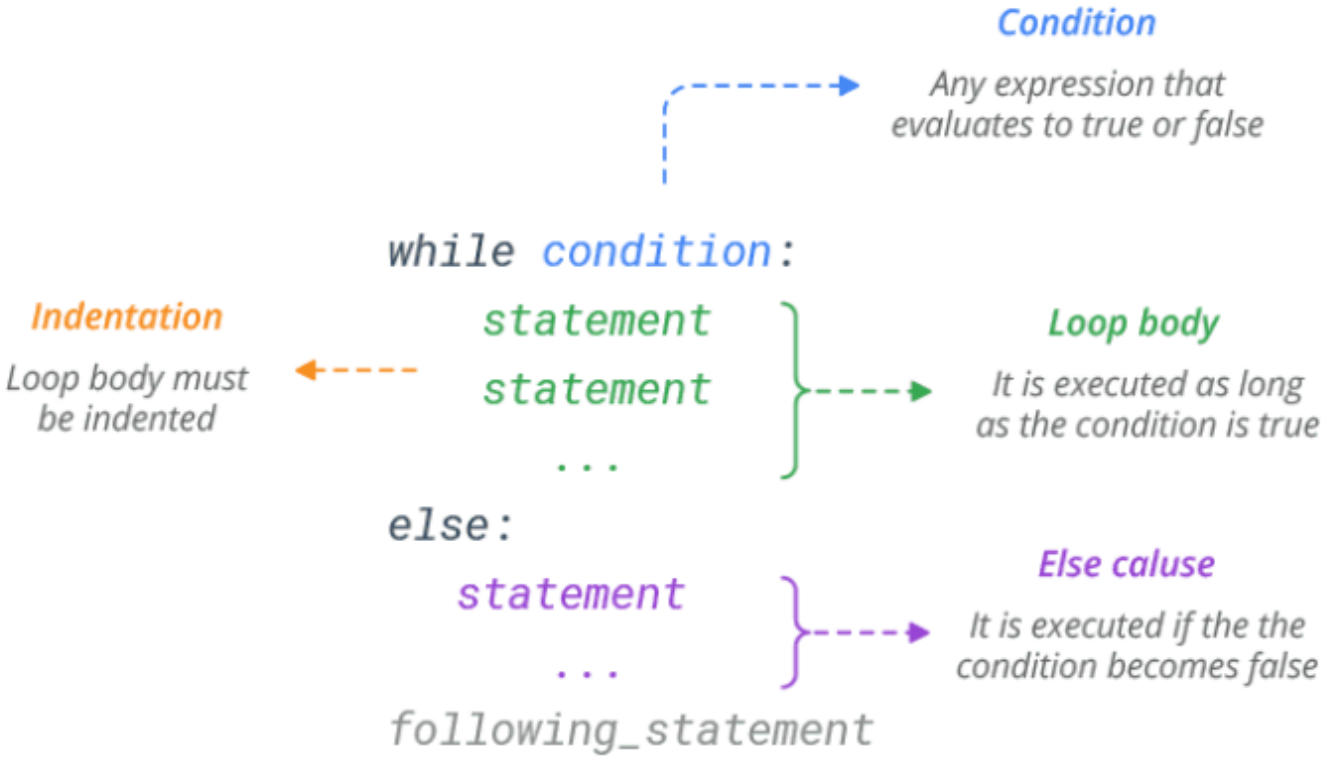
for loop (7)

Output:

```
Enter a number:  5
The factorial of 5 is 120.
```

while loop (1)

The `while` loop in Python repeatedly executes a block of code as long as the specified condition evaluates to `True`. Once the condition becomes `False`, the loop terminates.



while loop (2)

Write a Program to Find the Factorial of a Number using while loop.

```
num = int(input("Enter a number: "))

factorial = 1
i = 1

if num < 0:
  print("Factorial is not defined for negative numbers.")
elif num == 0:
  print("The factorial of 0 is 1.")
else:
  while i <= num:
    factorial *= i
    i += 1
  print(f"The factorial of {num} is {factorial}.")
```

while loop (3)

Output:

```
Enter a number:  5
The factorial of 5 is 120.
```

Infinite While Loop:

```
n = 0
while True:
  print(f"The latest value of n is {n}")
  n = n + 1
```

The statement `while True:` creates an infinite loop, meaning the loop will keep running unless explicitly stopped.

Table of Contents

The break Statement (1)

- The break statement is used to exit or break out of a loop prematurely, whether it is a for loop or a while loop.
- When the break statement is encountered, the program control is transferred to the first statement following the loop.

```
n = 0
while True:
  print(f"The latest value of n is {n}")
  n = n + 1
  if n > 5:  # Add a condition to stop the loop
    break
print("I am outside.")
```

The break Statement (2)

Output:

```
The latest value of n is 0
The latest value of n is 1
The latest value of n is 2
The latest value of n is 3
The latest value of n is 4
The latest value of n is 5
I am outside.
```

The continue Statement (1)

- The `continue` statement is used inside loops to skip the rest of the code in the current iteration and immediately move to the next iteration.
- It is typically used when you want to avoid some part of the loop under certain conditions, but still want to continue the loop for the next iteration.
- The `continue` statement can be used in both `for` and `while` loops.
- It is often used with an `if` condition to skip certain steps based on specific criteria.

Program 9: Skip Even Numbers and Print Odd Numbers Only

```
for i in range(1, 10):
  if i % 2 == 0:
    continue  # Skip even numbers
  print(i)  # Print odd numbers
```

The continue Statement (2)

Output:

```
1
3
5
7
9
```

The return Statement (1)

- The `return` statement is used to exit a function and optionally return a value to the caller.
- When the `return` statement is encountered, the function terminates immediately.
- The value provided after the `return` keyword is sent back to the function caller.
- If no value is provided, the function returns `None` by default.
- The `return` statement can only be used inside a function.

Program 10: Calculate the Square of a Number

```
def square(num):
    return num * num  # Returns the square of the number

result = square(5)  # Function call with argument 5
print(f"The square of 5 is: {result}")
```

The return Statement (2)

Output:

```
The square of 5 is:  25
```

Table of Contents

Syntax Error (1)

- A syntax error occurs when the code written in a program violates the language's grammar rules. Syntax is the structure or set of rules that defines how a valid program should be written. When Python encounters code that doesn't conform to these rules, it raises a `SyntaxError`.
- Syntax errors are typically detected during the parsing phase, before the code is executed. These errors prevent the program from running until they are fixed. They are the most common type of error that can easily corrected once identified.

Common Causes of Syntax Errors are:

- Incorrect Indentation
- Missing or Misplaced Colons
- Mismatched Parentheses, Brackets, or Braces
- Using Reserved Keywords as Variable Names
- Unclosed Strings

Syntax Error (2)

- Extra or Missing Commas
- Improper Use of Operators
- Incorrect Function Call Syntax

```
if x > 10
  print("x is greater than 10")
```

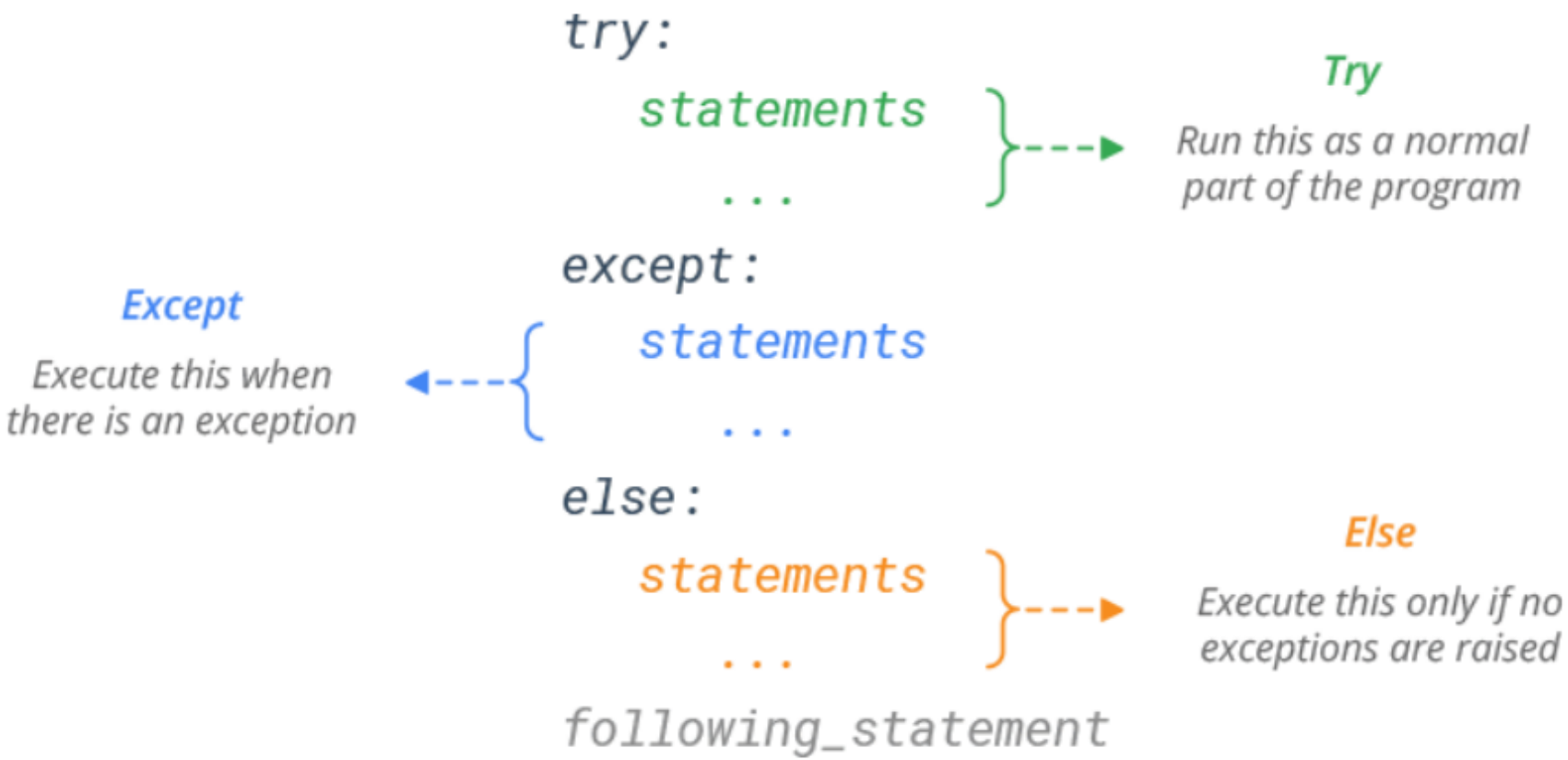
try... except (1)

Statements that can raise exceptions are wrapped inside the try block and the statements that handle the exception are written inside except block.

try... except (2)

```
try:
  # Code that may raise an exception
  num = int(input("Enter a number: "))  # This may raise
      ValueError if input is not an integer
  result = 10 / num  # This may raise ZeroDivisionError if
      num is 0
  print(f"The result of dividing 10 by {num} is: {result}")
except ValueError as e:
  # Handling invalid input (non-integer input)
  print(f"Invalid input! Please enter a valid number. Error:
      {e}")
except ZeroDivisionError as e:
  # Handling division by zero
  print(f"Error: Cannot divide by zero. Error: {e}")
```

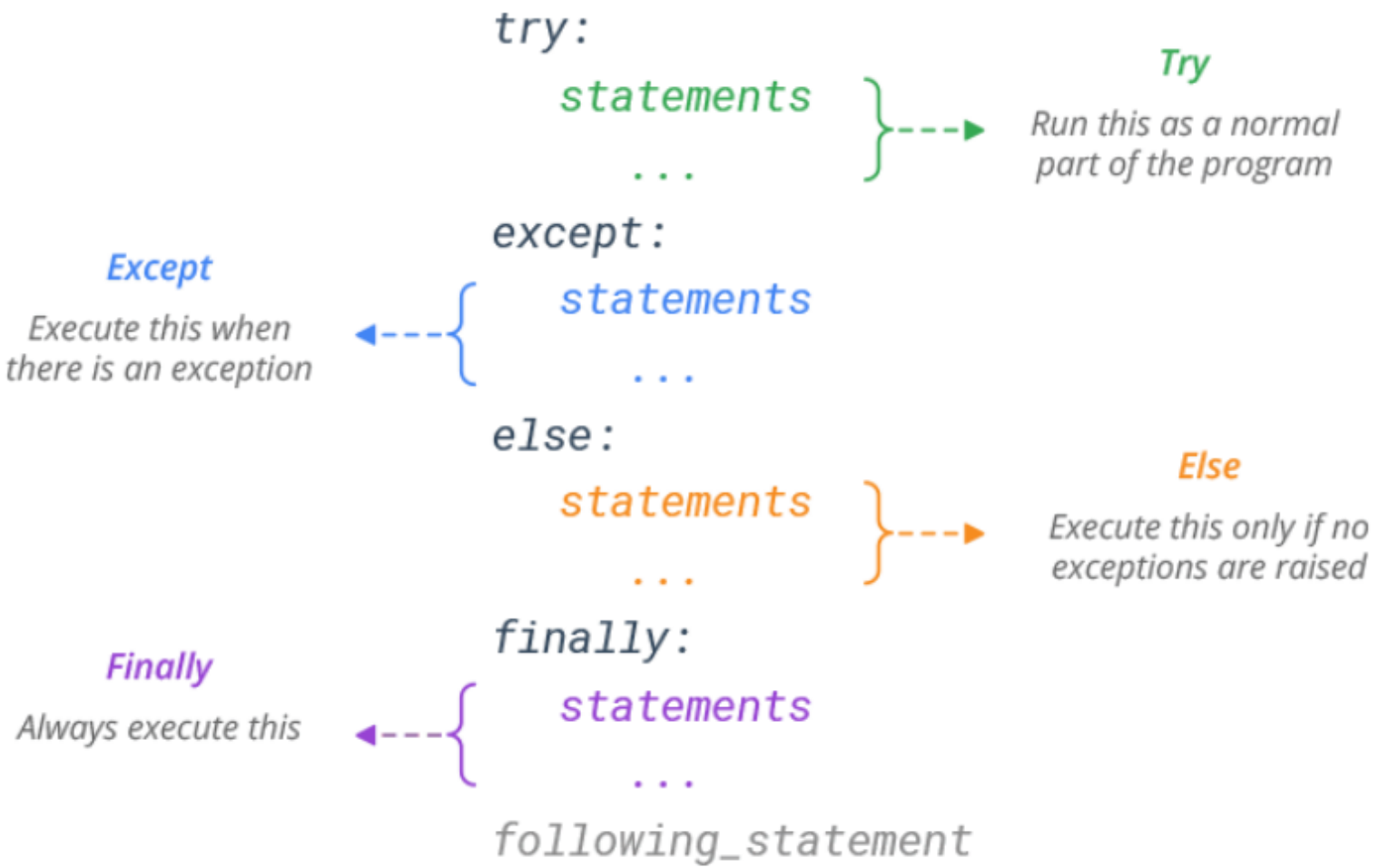
try ... except....else (1)



try ... except....else (2)

```
try:
  # Code that may raise an exception
  num = int(input("Enter a number: "))  # This may raise a
      ValueError if input is not an integer
  result = 10 / num  # This may raise a ZeroDivisionError if
       num is 0
except ValueError as e:
  # Handling invalid input (non-integer input)
  print(f"Invalid input! Please enter a valid number. Error:
       {e}")
except ZeroDivisionError as e:
  # Handling division by zero
  print(f"Error: Cannot divide by zero. Error: {e}")
else:
  # If no exception occurs, this block is executed
  print(f"The result of dividing 10 by {num} is: {result}")
```

try ... except....else...finally (1)



try ... except....else...finally (2)

```
try:
  # Code that may raise an exception
  num = int(input("Enter a number: "))  # This may raise
      ValueError if input is not an integer
  result = 10 / num  # This may raise ZeroDivisionError if
      num is 0
except ValueError as e:
  # Handling invalid input (non-integer input)
  print(f"Invalid input! Please enter a valid number. Error:
       {e}")
except ZeroDivisionError as e:
  # Handling division by zero
  print(f"Error: Cannot divide by zero. Error: {e}")
else:
  # If no exception occurs, this block is executed
  print(f"The result of dividing 10 by {num} is: {result}")
finally:
```

try ... except....else...finally (3)

```
# This block is always executed, regardless of whether an
    exception occurred or not
print("Execution completed.")
```

pass (1)

The pass statement is a null operation. It is a placeholder that does nothing when executed. It is commonly used in situations where a statement is syntactically required but you do not want to execute any code. This can be useful in situations like defining an empty class, function, or except block where you might not yet want to implement the functionality.

```
try:
    num = int(input("Enter a number: "))  # This may raise
        ValueError if input is not an integer
    result = 10 / num  # This may raise ZeroDivisionError if
        num is 0
except ValueError:
    # If a ValueError occurs (invalid input), do nothing
    pass
except ZeroDivisionError:
    # If a ZeroDivisionError occurs (dividing by zero), do
        nothing
    pass
```

pass (2)

```
else:
    # If no exception occurs, this block is executed
    print(f"The result of dividing 10 by {num} is: {result}")
finally:
    # This block always executes
    print("Execution completed.")
```