While Loops

Key Concepts:

- Functioning and Syntax of `while` loops
- Loop Counter
- Accumulator and Flag
- Nested Loops

Functioning and Syntax

Purpose: Repeat instructions until a condition changes.

Syntax:

```
while condition:
    # instructions
# code to execute after the loop
```

The instructions will be repeated as long as the condition is true. When the condition becomes false, the program continues with the code after the loop.

Example: Division of A by B

A program that asks for an integer A, then an integer B until B is non-zero, and then calculates the quotient of A by B.

```
a = int(input("Enter the value of A: "))
b = int(input("Enter the value of B: "))

while b == 0:
    b = int(input("B is zero! Try again: "))

print("A / B =", a // b)
```

Note: If B is non-zero on the first try, the `while` loop is not entered.

Avoiding Infinite Loops

If the condition of the `while` loop never becomes false, the program loops indefinitely:

Example 1:

```
n = 5
while n < 10:
    print("n is:", n)
    # Missing increment statement to avoid infinite loop
print("End")
```

Example 2:

```
while True:
    print("I am looping.")
    # No break condition to stop the loop
print("End")
```

Loop Counter

Purpose: Count how many times the loop is executed (or a number dependent on that).

Example:

```
b = int(input("Enter a non-zero integer b: "))
n = 0  # loop counter

while b == 0:
    n += 1
    b = int(input("Incorrect, try again: "))

print("Thank you, it took you", n, "extra tries.")
```

Important: Always initialize the counter.

Using the Counter in the Condition:

```
i = 0  # counter variable

while i < 10:
    print(2 ** i)  # 2 to the power of i
    i += 1  # increment the counter
print("End")
```

The step is the increment of the counter at each stage. Here, the step is 1.

Using Different Steps:

Step of 2:

```
i = 0  # counter variable

while i < 100:
    print(i)
    i += 2  # increment by 2
print("End")
```

Step of -1:

```
i = 10  # counter variable

while i > 0:
    print(i)
    i -= 1  # decrement by 1
print("End")
```

Accumulator Variable

Purpose: Store information about the values traversed, such as the sum:

```
i = 1
total_sum = 0  # initially, the sum is 0

while i <= 10:
    total_sum += i  # add each value of i to the sum (accumulation)
    i += 1  # never forget to update the counter

print("The sum of the first 10 integers is:", total_sum)
```

Initializing the Accumulator:

1. Don't forget to initialize it.
2. Use the neutral element of the operation:
    - For addition: 0 (because x + 0 = x)
    - For multiplication: 1 (because x * 1 = x)
    - For string concatenation: "" (empty string, because "" + "hello" = "hello")

Flag Variable

A boolean accumulator is called a flag.

Example: Read 10 integers and check that they are all odd:

```
i = 0
all_odd = True

while i < 10:
    x = int(input("Enter an integer: "))
    all_odd = all_odd and (x % 2 != 0)
    i += 1

if all_odd:
    print("All entered numbers are odd")
else:
    print("At least one entered number was not odd")
```

Initializing a Flag:

- For an AND operation: `True` (because `True and b` equals `b`)
- For an OR operation: `False` (because `False or b` equals `b`)

The `break` Keyword

Purpose: Exit the `while` loop immediately.

Example:

```
i = 1

while i < 100:
    if i % 2 == 0:
        print("*")
        break  # exit the loop
    i += 1

print("Incrementing i")
print("End")
```

The `continue` Keyword

Purpose: Immediately return to the beginning of the `while` loop, skipping the rest of the instructions in the loop.

Example:

```
i = 1

while i < 100:
    if i % 2 == 0:
        print("*")
        continue  # skip the rest of the loop
    i += 1

print("Incrementing i")
print("End")
```

Disadvantages of `break` and `continue`:

- Code can be harder to read/analyze with multiple levels of nesting and/or long instructions within the `while` loop.
- Not always available in other programming languages.

Alternative to `break`:

```
stop = False

while not stop and condition:
    # instructions
    if ...:
        stop = True
    if not stop:
        # other instructions
```

Nested Loops

A `while` loop can contain another `while` loop as an instruction.

Example: What is the output of this program?

```
i = 1

while i <= 3:
    j = 1
    while j <= 2:
        print(i, ", ", j)
        j += 1
    i += 1
```

Output:

```
1, 1
1, 2
2, 1
2, 2
3, 1
3, 2
```

Application Example:

Write a program that displays an `n x n` square of the character `*`. The user chooses the side `n` of the square.

Example Output:

```
Enter the value of n: 5
*****
*****
*****
*****
*****
```

Solution:

```
n = int(input("Enter the value of n: "))

num_row = 0  # line counter

while num_row < n:
    num_col = 0  # star counter for the line
    while num_col < n:
        print("*", end="")  # use end to stay on the same line
        num_col += 1
    print()  # new line
    num_row += 1  # move to the next line
```