While Loop Class Notes

Class Notes: Python `while` Loops

1. Introduction to `while` Loops

- A `while` **loop** executes a block of code repeatedly **as long as a condition remains** `True` .
- Commonly used when:
  1. The number of iterations is **not** known in advance (e.g., reading user input until a certain condition is met).
  2. You need **real-time condition checks** to decide whether to continue.

Contrast with `for` Loops

- `for` loops: Usually preferred when you know the number of iterations (e.g., 5 times, or iterate over a list).
- `while` loops: Ideal if you only know you want to keep running “while” something is true.

2. Basic Syntax

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

- **condition**: An expression that evaluates to `True` or `False` .
- The loop continues as long as `condition` is `True` .
- Once `condition` is `False` , the loop stops.

Example: Counting Up

```
x = 5
i = 1

while i <= x:
```

```
    print(i)
    i += 1  # increment i to avoid infinite loop
```

- Prints numbers from 1 to 5.
- If you forget to increment `i`, the loop may run forever (`infinite loop`).

3. Using `while True`

Sometimes you may use an **infinite while loop** intentionally, then break out when a condition is met:

```
while True:
    user_input = int(input("Enter a number: "))
    if user_input == 1:
        break
    else:
        print("Hello World")
```

- This pattern is common if you don't know how many times the user will input data.
- Use `break` to exit once your terminating condition is satisfied.

4. `break` and `continue` in `while` Loops

4.1. `break`

- Exits the loop **immediately**, skipping all remaining iterations.
- Often used after confirming a certain condition is met.

Example:

```
i = 1
while i <= 10:
    if i == 5:
        break
    print(i)
    i += 1
```

- The loop ends entirely when `i` is 5.

4.2. `continue`

- Skips **just the current iteration** and continues with the next.

- Typically used if you want to skip processing a particular case.

Example:

```
i = 1
while i <= 10:
    i += 1
    if i == 5:
        continue
    print(i)
```

- When `i` is 5, that iteration is skipped, and 5 is not printed.

5. The `else` Clause in `while` Loops

- Python allows an `else` block after a `while` loop.
- This `else` **only executes if the loop finishes normally** (i.e., it did not encounter a `break`).

```
i = 1
while i <= 10:
    i += 1
    if i == 5:
        continue
    print(i)
else:
    print("Finished")
```

- **If** a `break` statement occurs, the `else` will not run.
- In this example, since `break` is never called, the `else` block (“Finished”) executes once the loop condition (`i <= 10`) fails.

6. Practical Scenarios

1. **User Input**
   - Continuously read data from a user until they type a certain value (e.g., `0` or `1`).
2. **Waiting for a Condition**
   - In real-time systems (e.g., sensor data) where you only stop once a threshold is exceeded.
3. **Menu Systems**
   - Show a menu and prompt the user until “Exit” is selected.

7. Common Pitfalls

4. **Forgetting to Update the Loop Variable**
   - Can cause an **infinite loop**, e.g., `while i < 5: print(i)`.
5. **Misusing `break`**
   - Breaking too soon or failing to break when needed can lead to logic errors.
6. **Complex Logic**
   - While loops can quickly become tricky if you mix multiple breaks, continues, and nested conditions. Keep it simple where possible or refactor into smaller functions.

8. Summary

- `while` **loops** run code while a certain condition is true, making them powerful for indefinite iteration.
- `break` ends the loop instantly; `continue` skips the rest of the current iteration.
- An `else` clause can run if the loop concludes normally (no break).
- Always ensure your loop condition eventually becomes `False`, or use a `break` to avoid infinite loops.

With these fundamentals, you can effectively control program flow in scenarios where the exact number of iterations is unknown or dependent on external factors (user input, real-time data, etc.).