مكتب

Quick Reference Notes

Week 11: Smart Loops - Loops with Conditions

7th Grade Computer Science

1 Introduction

1.1 This Week's Big Question

How can we make loops that think? Instead of blindly repeating every single time, what if our loops could skip certain items, stop early when they find what they're looking for, or make smart decisions while repeating? Just like a teacher checking homework who only marks correct answers, we'll create intelligent loops!

Prerequisites

Before starting this week, you should be comfortable with:

- Creating `if/else` and elif statements (Week 8)
- Writing for loops with `range()` (Week 9)
- Understanding nested conditions and complex boolean logic (Week 8)
- Creating nested loops and understanding their flow (Week 10)

1.2 What You Already Know

You can write programs that make decisions using if statements. You can create loops that repeat a specific number of times. You understand how to use boolean expressions to test conditions. You've even put loops inside other loops to create 2D patterns!

1.3 What You'll Be Able to Do

By the end of this week, you'll:

- Filter data inside loops *(like selectively printing only even numbers)*
- Stop loops early when you find what you're looking for
- Create programs that validate user input using loops
- Build interactive menu systems using for loops
- Combine multiple programming concepts into sophisticated programs

2 VIDEO 1: Combining Loops with Conditions

2.1 Making Loops Selective

So far, our loops process every single item the same way. But real programs need to be selective - like how you might look through your backpack for only your math notebook, ignoring everything else.
Let's See It In Action

Print only the even numbers from 1 to 20:

```
print("Even numbers from 1 to 20:")
for num in range(1, 21):
    if num % 2 == 0:
        print(num, end=" ")

# Output:
# Even numbers from 1 to 20:
# 2 4 6 8 10 12 14 16 18 20
```

Breaking It Down

The magic happens when we put an if statement inside a for loop:

- The loop visits every number from 1 to 20
- The if statement acts as a "filter"
- Only numbers that pass the test *(divisible by 2)* get printed
- Numbers that fail the test are skipped

2.2 Example: Counting Matches

Count how many numbers between 1 and 30 are divisible by both 3 and 5:

```
count = 0

for num in range(1, 31):
    if num % 3 == 0:
        count = count + 1
        print(f"{num} is divisible by 3")

print(f"\nTotal count: {count}")

# Output shows: 3, 6, 9, 12, 15, 18, 21, 24, 27, 30
# Total count: 10
```

How It Works:

- The loop checks each number from 1 to 30
- If the number is divisible by 3, it increments the count
- At the end, it prints how many numbers matched the condition

Tip

Recall: the `range(start, end)` function generates numbers starting from `start` up to (but not including) `end`. So `range(2, 11)` gives you numbers 2 through 10.

Quick Check

What numbers will this code print?

```
for num in range(3, 13):
    if num > 8:
        print(num)
```

3 VIDEO 2: Smart Loop Termination with break

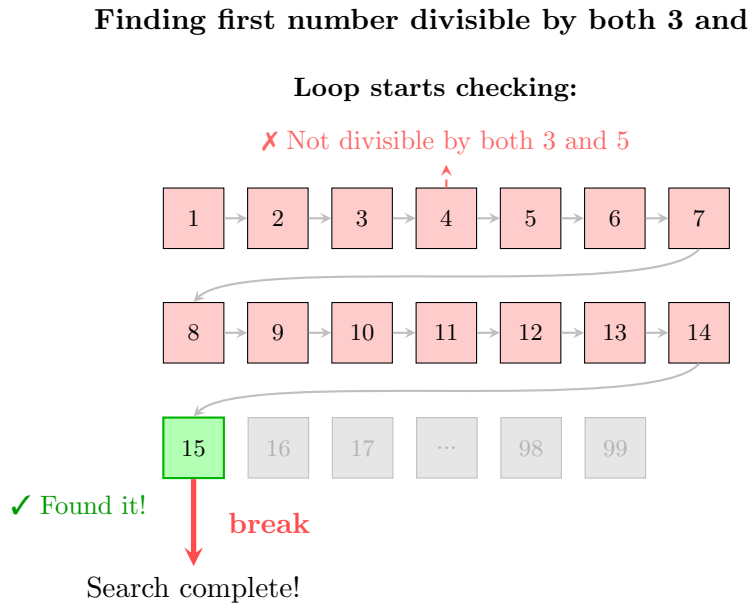
3.1 Stopping Early When You Find It

Imagine looking for your favorite pencil in a box of 100 pencils. Once you find it, do you keep looking? Of course not! The `break` statement lets our loops be just as smart.

3.2 Let's See It In Action

```
# Find the first number divisible by both 3 and 5
print("Searching for a number divisible by both 3 and 5...")

for num in range(1, 100):
    if num % 3 == 0 and num % 5 == 0:
        print(f"Found it! {num} is divisible by both 3 and 5")
        break  # Exit the loop immediately

print("Search complete!")

# Output:
# Searching for a number divisible by both 3 and 5...
# Found it! 15 is divisible by both 3 and 5
# Search complete!
```

3.3 How break Works



3.4 Practical Example: Finding Perfect Squares

A **perfect square** is a number that you get when you multiply a whole number by itself!

```
# Find the first perfect square greater than 50
print("Searching for first perfect square greater than 50...")

for num in range(1, 20):
    square = num * num
    print(f"Checking: {num} squared = {square}")

    if square > 50:
        print(f"\nFound it! {square} is the first perfect square > 50")
        print(f"It's {num} squared")
        break

# Output:
# Checking: 1 squared = 1
# Checking: 2 squared = 4
# ... (continues) ...
# Checking: 7 squared = 49
# Checking: 8 squared = 64
#
# Found it! 64 is the first perfect square > 50
# It's 8 squared
```

Common Error

Common break Mistakes:

- Forgetting `break` in search loops → Loop continues unnecessarily
- Code after `break` in same block → That code never runs!
- Using `break` in nested loops → Only exits the innermost loop
- No flag variable → Can't tell if loop ended naturally or with break

Quick Check

What number will this print?

```
for x in range(10, 20):
    if x % 4 == 0:
        print(x)
        break
```

4 VIDEO 3: Building Complex Logic Flows

4.1 Combining Everything We Know

Real programs combine multiple concepts. Let's see how loops, conditions, and break work together to create sophisticated programs that can handle complex tasks.

4.2 Counting Specific Items

```python
# Count how many numbers from 1-30 are divisible by both 2 and 3
count = 0
print("Numbers divisible by both 2 and 3:")

for num in range(1, 31):
    if num % 2 == 0 and num % 3 == 0:
        print(num, end=" ")
        count = count + 1

print(f"\n\nTotal found: {count}")

# Output:
# Numbers divisible by both 2 and 3:
# 6 12 18 24 30
#
# Total found: 5
```

4.3 Interactive Menu System

```python
# Calculator menu using for loop
print("=== Simple Calculator ===")
exit_program = False

# We'll give the user up to 100 operations
for operation in range(100):
    if exit_program:
        break

    print("\n1. Add")
    print("2. Subtract")
    print("3. Multiply")
    print("4. Exit")

    choice = input("Enter choice (1-4): ")

    if choice == "1":
        num1 = int(input("First number: "))
        num2 = int(input("Second number: "))
        print(f"Result: {num1} + {num2} = {num1 + num2}")
    elif choice == "2":
        num1 = int(input("First number: "))
        num2 = int(input("Second number: "))
        print(f"Result: {num1} - {num2} = {num1 - num2}")
    elif choice == "3":
        num1 = int(input("First number: "))
        num2 = int(input("Second number: "))
        print(f"Result: {num1} * {num2} = {num1 * num2}")
    elif choice == "4":
        print("Thank you for using the calculator!")
        exit_program = True
    else:
        print("Invalid choice! Please try again.")
```

4.4 Input Validation with Multiple Criteria

```python
# Get student age - must be reasonable for Grade 7
valid_age = False

for attempt in range(3):
    age = int(input("Enter your age: "))

    if age < 10:
        print("Too young for Grade 7! Please check.")
    elif age > 16:
        print("Too old for Grade 7! Please check.")
    elif age >= 11 and age <= 14:
        print(f"Perfect! Age {age} recorded.")
        valid_age = True
        break
    else:
        print(f"Age {age} is unusual for Grade 7, but accepted.")
        valid_age = True
        break

if not valid_age:
    print("Unable to verify age after 3 attempts.")
```

This example shows validation with multiple conditions and different responses based on the input range.

Rule

Complex Logic Patterns:

1. **Menu with exit**: Use flag variable + break
2. **Input validation**: Use for loop with limited attempts
3. **Search and process**: Combine if conditions with break
4. **Count and filter**: Use counter variable with conditions
5. **Multiple exit points**: Use flag variables to track state

Quick Check

Compare these two ways to exit the menu:

1. Using `break` directly in choice 4
2. Using an `exit_program` flag variable

Can you think of a situation where the flag approach would be better?"

5 VIDEO 4: Practical Applications

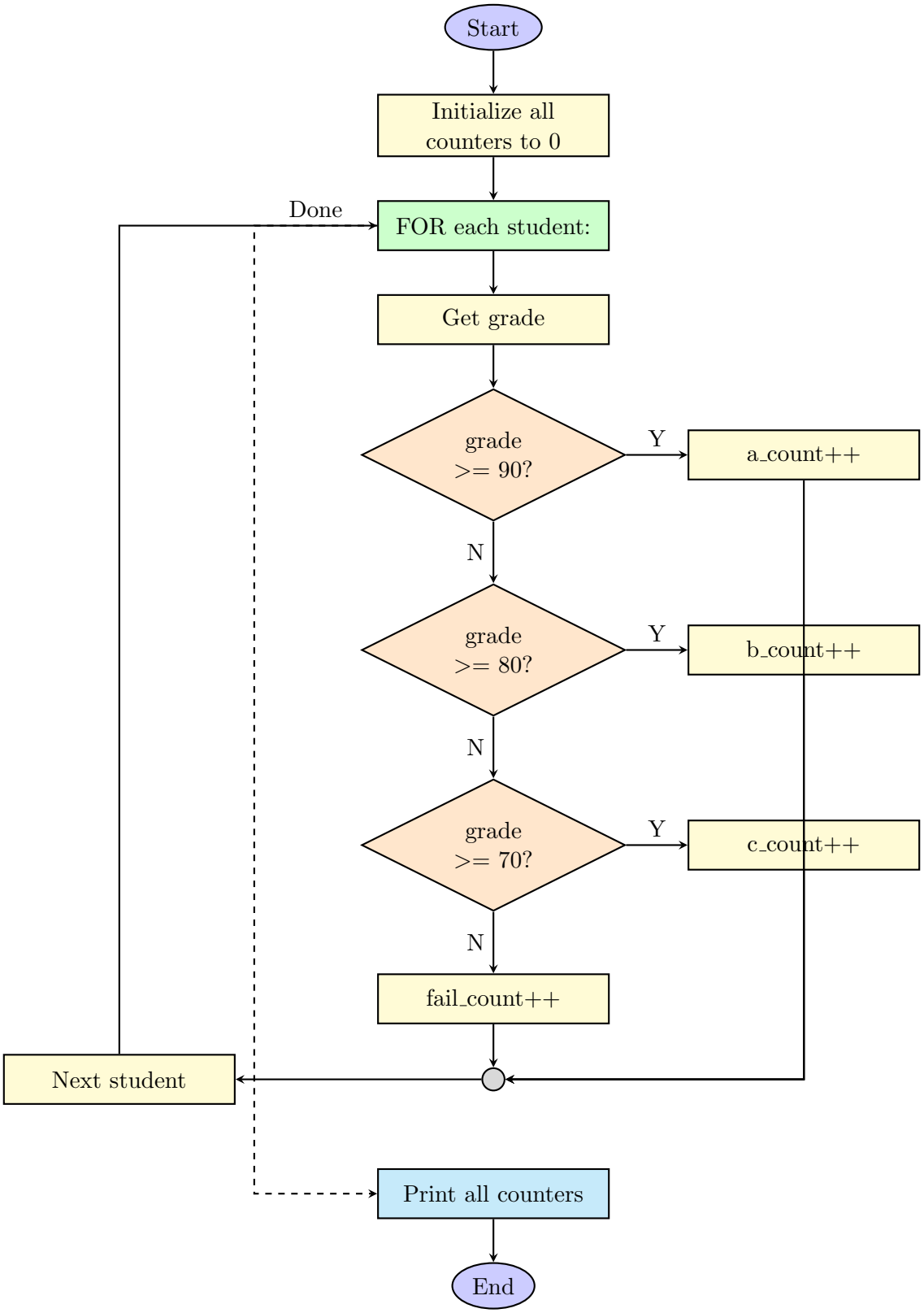
5.1 Number Guessing Game

Let's build a complete game using everything we've learned:

```
# Two-player guessing game
print("=== Number Guessing Game ===")
print("Player 1: Pick a number between 1 and 20")
print("Player 2: Try to guess it!")

# Player 1 picks the secret number
secret = int(input("\nPlayer 1, enter your secret number (1-20): "))
print("\n" * 20)  # Clear screen with blank lines

# Player 2 gets 5 guesses
max_guesses = 5
found = False

print("Player 2: I'm thinking of a number between 1 and 20!")
print(f"You have {max_guesses} guesses.")

for guess_num in range(max_guesses):
    remaining = max_guesses - guess_num
    guess = int(input(f"\nGuess {guess_num + 1} ({remaining} left): "))

    if guess == secret:
        print("*** Congratulations! You got it! ***")
        found = True
        break
    elif guess < secret:
        print("Too low! Try higher.")
    else:
        print("Too high! Try lower.")

if not found:
    print(f"\nSorry! The number was {secret}")
    print("Better luck next time!")
```

5.2 Grade Analysis Program

Let's analyze student grades using a loop that counts how many students fall into different grade ranges:



```python
# Analyze grades for a class - find highest, lowest, and count grades
num_students = int(input("How many students? "))
highest = -1  # Start with impossible low value
lowest = 101  # Start with impossible high value
total = 0

for student in range(num_students):
    grade = int(input(f"Enter grade for student {student + 1}: "))

    # Update highest and lowest
    if grade > highest:
        highest = grade
    if grade < lowest:
        lowest = grade

    # Add to total for average
    total = total + grade

    # Print grade category
    if grade >= 90:
        print("Grade: A - Excellent!")
    elif grade >= 80:
        print("Grade: B - Good job!")
    elif grade >= 70:
        print("Grade: C - Satisfactory")
    else:
        print("Grade: F - Needs improvement")

print("\n=== Class Statistics ===")
print(f"Highest grade: {highest}")
print(f"Lowest grade: {lowest}")
```

Tip

Building Complex Programs:

- Start simple - get basic loop working first
- Add conditions one at a time
- Test after each addition
- Use meaningful variable names
- Add comments to explain complex logic

Quick Check

How would you modify the grade analysis program to also calculate the class average?

6 VIDEO 5: Summary

6.1 The Power of Smart Loops

This week, we've transformed simple counting loops into intelligent programs that can filter, search, validate, and interact with users. You're not just repeating blindly anymore - you're teaching the computer to think while it loops!

6.2 Key Concepts We Mastered

Pattern	Example	Key Feature
Filter	Even numbers only	Process selective items
Find First	First perfect square > 50	Exit when found
Count Matches	Numbers divisible by 2 AND 3	Track occurrences
Multi-Validation	Age range checking	Different responses per range
Menu System	Calculator choices	Continuous operation
Track Min/Max	Grade statistics	Update tracking variables
Guided Search	Number guessing	Feedback directs search

6.3 From Simple to Smart

Look at your progression:

- **Week 9:** Basic loops that count
- **Week 10:** Nested loops for patterns
- **Week 11:** Intelligent loops that think!

★ Landmark Moment

You've unlocked the secret to smart repetition! Your loops can now make decisions, stop when needed, and interact intelligently with users. This is a massive achievement - you're combining multiple concepts to create real, useful programs. Every search engine, every game, every app uses these exact patterns. You're thinking like a real programmer now!

6.4 Reflection

Think about your learning journey this week:

1. We combined loops with conditions to create filters. What other things in real life work like filters?
2. The break statement saves unnecessary work. Can you think of three real-world situations where you "break" out of a repetitive task?
3. Menu systems let users control programs. What makes a menu user-friendly?
4. Input validation prevents errors. Why is it important to validate user input before using it?
5. You can now build interactive programs! What program would you like to create using these tools?
6. When debugging loops with conditions, what helps you track the program flow?
7. How has combining concepts (loops + conditions + break) made your programs more powerful?

Next week, we'll explore systematic debugging strategies to help you find and fix errors like a professional programmer!