Minor in AI

Python Programming Basics

1 Case Study

Imagine you're at a casino, flipping a coin repeatedly. You're trying to determine if the coin is fair or biased. How would you keep track of the number of heads and tails? This real-world scenario introduces us to the concept of loops and conditional statements in programming.

2 Introduction

Figure 1: Evolution from tapes to disks.

The evolution of storage devices, from magnetic tapes to modern disks, drawing an analogy to the development of programming languages. Just as storage technology progressed from bulky cassettes to compact disks and eventually to high-capacity hard drives, programming languages have evolved to become more efficient and user-friendly. This evolution is driven by the need to store and process increasing amounts of data, much like how programming languages have developed to handle more complex tasks. This comparison highlights the importance of understanding the fundamental principles behind programming languages, as it allows us to appreciate the reasons behind certain design choices and potentially innovate new solutions.

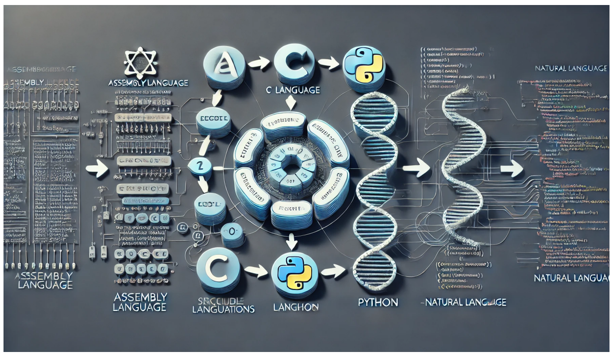

Figure 2: Conceptual illustration of Evolution from Assembly language to C to Python to natural human language.

3 Loops and Ifs

3.1 Problem Definition

The problem we're addressing is how to efficiently count the number of heads and tails in a series of coin flips. This introduces us to several key programming concepts:

1. Data storage (lists)

2. Loops (for and while)
3. Conditional statements (if-else)
4. Variable manipulation

3.2 Solution: Python Code

Let's look at a Python solution to this problem:

```python
# Define the list of coin flips
flips = ['H', 'T', 'T', 'H', 'H', 'T']

# Initialize counters for heads and tails
head_count = 0
tail_count = 0

# Iterate through each flip in the list
for f in flips:
    if f == 'H':  # If the flip is a head, increment head_count
        head_count += 1
    elif f == 'T':  # If the flip is a tail, increment tail_count
        tail_count += 1

# Print the final count of heads and tails
print(f"Head count: {head_count}")
print(f"Tail count: {tail_count}")
```

This code demonstrates:
- List usage to store coin flip results
- A 'for loop' to iterate through the list
- Conditional statements to check for heads or tails
- Incrementing counters

3.3 Alternative Solution: While Loop

We can also solve this problem using a while loop:

```python
# Define the list of coin flips
flips = ['H', 'T', 'T', 'H', 'H', 'T']

# Initialize counters for heads and tails
head_count = 0
tail_count = 0

# Initialize loop index and total number of flips
i = 0
n = len(flips)  # Get the length of the list

# Loop through each element in the list using a while loop
while i < n:
    if flips[i] == 'H':  # If the flip is a head, increment head_count
        head_count += 1
    elif flips[i] == 'T':  # If the flip is a tail, increment tail_count
        tail_count += 1
    i += 1  # Move to the next index

# Print the final count of heads and tails
print(f"Head count: {head_count}")
print(f"Tail count: {tail_count}")
```

This version introduces:
- The concept of indexing in lists
- Use of a 'while loop' with a counter
- Pre-computing the length of the list for efficiency

4 Bitwise Operators

In addition to basic arithmetic and logical operations, Python supports bitwise operators. These operators manipulate individual bits of integers, providing efficient low-level operations that are widely used in various applications.

4.1 Types of Bitwise Operators

The bitwise operators are:

- **AND** : Performs a bitwise AND operation. Returns 1 if both corresponding bits are 1, otherwise 0.

  $$\text{Example: } 1010_2 \,\&\, 1100_2 = 1000_2$$

- **OR** : Performs a bitwise OR operation. Returns 1 if at least one of the corresponding bits is 1, otherwise 0.

  $$\text{Example: } 1010_2 \,|\, 1100_2 = 1110_2$$

- **NOT** : Performs a bitwise negation (1's complement). Inverts all bits, changing 1s to 0s and 0s to 1s.

  $$\text{Example (8-bit representation): } \sim 1010_2 = 0101_2$$

- **Left Shift** : Shifts the bits to the left by a specified number of positions, filling with 0s on the right. This operation is equivalent to multiplying by 2^n, where n is the number of shifts.

  $$\text{Example: } 1010_2 << 2 = 101000_2$$

- **Right Shift** : Shifts the bits to the right by a specified number of positions. If it is a logical right shift, it fills the leftmost bits with 0s. If it is an arithmetic right shift, it fills with the sign bit (for negative numbers). Equivalent to integer division by 2^n.

  $$\text{Example: } 1010_2 >> 2 = 10_2$$

- **1's Complement**: Inverts all bits of a number (same as bitwise NOT). Used in some old systems for representing negative numbers.

  $$\text{Example (8-bit representation): 1's complement of } 00001010_2 = 11110101_2$$

- **2's Complement**: Obtained by taking the 1's complement of a number and adding 1. It is the standard way to represent negative numbers in binary systems.

  $$\text{Example (8-bit representation): 2's complement of } 00001010_2$$

  $$1's \text{ complement: } 11110101_2$$

  $$\text{Adding 1: } 11110110_2$$

4.2 Example: Bitwise AND

Let's look at how the bitwise AND operator works:

```
a = 12 # 1100 in binary
b = 25 # 11001 in binary
result = a & b # 1100 & 11001 = 01100 (8 in decimal)
print(f"Result of {a} & {b} = {result}")
```

Step-by-step Explanation:

- Convert decimal numbers to binary:

$$12_{10} = 1100_2, \quad 25_{10} = 11001_2$$

- Perform bitwise AND on corresponding bits:

$$01100_2 \& 11001_2 = 01000_2$$

- Convert the binary result back to decimal:

$$01000_2 = 8_{10}$$

- Print the result: `Result of 12 & 25 = 8`

4.3 Applications of Bitwise Operators in AI

Bitwise operators are commonly used in AI and machine learning in the following ways:

- **Feature Engineering:** Used to optimize storage and processing of binary features in datasets.
- **Efficient Computations:** Speeds up neural network operations, such as activation functions and matrix transformations.
- **Genetic Algorithms:** Helps implement mutation and crossover operations efficiently.
- **Image Processing:** Used in image masking, thresholding, and edge detection.
- **Cryptography:** Assists in encryption and hashing techniques used in secure AI applications.

5 Key Takeaways

This section covers fundamental Python concepts, including Boolean operations with logical operators (AND, OR, NOT) and bitwise operators, highlighting the difference between the two. It explores Python's core data types such as integers, floats, characters, strings, and Booleans, along with conditional statements like the if-else structure. Looping constructs, including for and while loops, are discussed with emphasis on list operations—creating lists, accessing elements, and variable management such as initialization and incrementing. The importance of code visualization using Python Tutor is demonstrated by analyzing memory frames and variable states during execution. Practical examples include a coin toss simulation program that counts heads and tails using loops and an exploration of one's complement and two's complement concepts. Additionally, programming best practices are introduced, such as loop optimization techniques like pre-computing list lengths, along with an overview of the significance of understanding fundamental principles in programming. The section also briefly introduces the concept of loop rolling and unrolling for efficient execution.