Counting in Base 10

Theodore M. Savich

April 4, 2025

1 Diagonalizing the Count

1.1 Sublation in Counting: From Tallies to Base Systems

Counting is not merely an accumulation of marks – it is a process that both *preserves* and *transforms* prior determinations. In Hegelian terms, this movement is called *sublation* (Aufhebung), the simultaneous *negation*, *preservation*, and *uplift* of what came before. In mathematical practice, sublation is most clearly seen in the way base systems reorganize quantities into new structural units.

Consider a simple act of tally counting. If one were to count to nine using tally marks, the representation would appear as:

$$|||||||||$$

Each tally stands independently as a discrete marker of a counted object that mirrors the "world of ones" reflected in von Neumann ordinals. They could just go on and on, accumulating indefinitely. While it is more normal to represent a transformation at 5 units, let us instead live in base ten. When ten is reached, the representation undergoes an important transformation:

$$\cancel{|||||||||}$$

The previous nine marks are not erased. They are not 'gone.' But they are *negated* and *uplifted* into a new structural form. Out of the many ones, there is now one ten. This is a mathematical instance of sublation. The prior elements are not discarded. They are reorganized in a higher-level composition. The transition from loose tallies to a single "ten" does not merely introduce a new symbol; it alters how the prior marks are understood. They are still 'present,' but they no longer function as isolated entities.

So, using base systems involves "two views" of a number - but under the hood is very basic version of a diagonalizing function, δ, that lets an element reference the whole system it's part of. Ten loose ones is a "many", one 10 is a "one". Diagonalization is, therefore, a way of thinking about the problem of the one and the many.

2 Understanding the Recursive Nature of Counting

Counting in base 10 involves incrementing digits and managing composition across multiple place values:

- **Units (Ones):** $10^0 = 1$
- **Tens:** $10^1 = 10$

- **Hundreds:** $10^2 = 100$
- **Thousands:** $10^3 = 1,000$, etc.

The recursive process for counting follows these steps:

1. Increment the units digit.
2. If the units digit reaches 10, reset it to 0 and increment the tens digit.
3. Repeat this process recursively for higher place values as needed.

This recursive nature allows for counting indefinitely by reusing the same increment and composition logic for each digit.

3 Why Use a Pushdown Automaton (PDA)?

A Pushdown Automaton (PDA) is suitable for modeling recursive counting due to its ability to use a stack for memory. Here's why:

- **Finite State Automaton (FSA):** Lacks the memory to handle arbitrary-length counts and composition.
- **Pushdown Automaton (PDA):** Uses a stack to provide additional memory, enabling nested operations like composition in counting.
- **Turing Machine:** While capable, it is more complex than needed for this task.

A PDA's stack can represent digit states and manage composition recursively, making it an appropriate choice.

4 Designing a PDA for Three-Digit Base-10 Counting (0-999)

While the unbounded recursive nature of counting presents challenges for standard PDA models, we can successfully design a PDA to handle counting within a fixed, practical range. This section details a Deterministic Pushdown Automaton (DPDA) capable of counting from 0 to 999, demonstrating how the stack and finite state control can manage multi-digit carries. This approach avoids the theoretical issues of infinite states or alphabets required by some conceptual models, providing a formally sound automaton for a three-digit counter.

4.1 Simulating Three Digits on One Stack

We represent the three place values (Hundreds, Tens, and Units) using distinct symbols on the PDA's single stack:

- **Units Digit Symbols:** $U_0, U_1, \ldots, U_9$
- **Tens Digit Symbols:** $T_0, T_1, \ldots, T_9$
- **Hundreds Digit Symbols:** $H_0, H_1, \ldots, H_9$

A number, represented conventionally as XYZ (X=Hundreds, Y=Tens, Z=Units), will be stored on the stack with the units digit on top. The stack configuration will be $(\#, H_X, T_Y, U_Z)$, where $\#$ is the bottom marker. For example, the number 123 would be represented by the stack $(\#, H_1, T_2, U_3)$.

4.2 Components of the Three-Digit PDA

The PDA is defined by the 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, \#, F)$:

- **States (Q):** A finite set of states manages the counting and multi-level carry logic:
  1. q_{start}: The initial state for setup.
  2. q_{idle}: The main state where the PDA resides when holding a valid count (0-999). This is the accepting state.
  3. $q_{\text{inc\_tens}}$: Intermediate state entered when the units digit rolls over ($U_9 \to U_0$), signaling the need to process the tens digit via an epsilon transition.
  4. $q_{\text{inc\_hundreds}}$: Intermediate state entered when the tens digit rolls over ($T_9 \to T_0$), signaling the need to process the hundreds digit via an epsilon transition.
  5. q_{halt}: A non-accepting state entered when an attempt is made to increment the count beyond 999 (hundreds digit rollover).
- **Input Alphabet (Σ):** Contains a symbol representing one unit to be counted, plus the empty string ϵ for internal transitions.

$$\Sigma = \{\text{tick}, \epsilon\}$$

- **Stack Alphabet (Γ):** Includes the bottom marker and symbols for each digit in each place value.

$$\Gamma = \{\#, H_0, \ldots, H_9, T_0, \ldots, T_9, U_0, \ldots, U_9\}$$

- **Transition Function (δ):** Defined formally in Section 8.
- **Initial State (q_0):** q_{start}.
- **Initial Stack Symbol (Z_0):** #. (Implicitly placed on the stack at start).
- **Final States (F):** Only the state representing a valid count within the range is accepting.

$$F = \{q_{\text{idle}}\}$$

4.3 Automaton Behavior

The three-digit counter operates as follows:

1. **Initialization:** Start in q_{start}. On an epsilon transition seeing #, push H_0, T_0, U_0 onto the stack (representing 0) and transition to q_{idle}. Stack: $(\#, H_0, T_0, U_0)$.
2. **Counting (Units Increment):** In state q_{idle}, read a 'tick' input.
   - If the top symbol is U_n where $n < 9$, pop U_n, push U_{n+1}, and remain in q_{idle}.
   - If the top symbol is U_9, pop U_9, push nothing. Transition to $q_{\text{inc\_tens}}$ to handle the carry to the tens place. The T_Y symbol is now exposed on top.
3. **Carry Handling (Tens Increment):** In state $q_{\text{inc\_tens}}$, perform an epsilon transition based on the exposed tens digit T_m:

- If the top symbol is T_m where $m < 9$, pop T_m. Push T_{m+1}, then push U_0. Transition back to q_{idle}. The carry is complete. Stack: $(\#, H_X, T_{m+1}, U_0)$.
- If the top symbol is T_9, pop T_9, push nothing. Transition to $q_{\text{inc\_hundreds}}$ to handle the carry to the hundreds place. The H_X symbol is now exposed on top.

4. **Carry Handling (Hundreds Increment):** In state $q_{\text{inc\_hundreds}}$, perform an epsilon transition based on the exposed hundreds digit H_k:

   - If the top symbol is H_k where $k < 9$, pop H_k. Push H_{k+1}, then push T_0, then push U_0. Transition back to q_{idle}. The carry is complete. Stack: $(\#, H_{k+1}, T_0, U_0)$.
   - If the top symbol is H_9 (representing an attempt to increment 999), pop H_9. Push H_0, then push T_0, then push U_0. Transition to the non-accepting state q_{halt}. Stack: $(\#, H_0, T_0, U_0)$.

5. **Halt State:** Once in q_{halt}, no further transitions are defined. The machine halts and implicitly rejects any further input, indicating overflow beyond 999.

5 State Diagram for Three-Digit Counter

The diagram illustrates the states and key transitions. Multi-symbol stack operations are abbreviated (e.g., $T_m \to U_0, T_{m+1}$ means pop T_m, push T_{m+1}, push U_0).

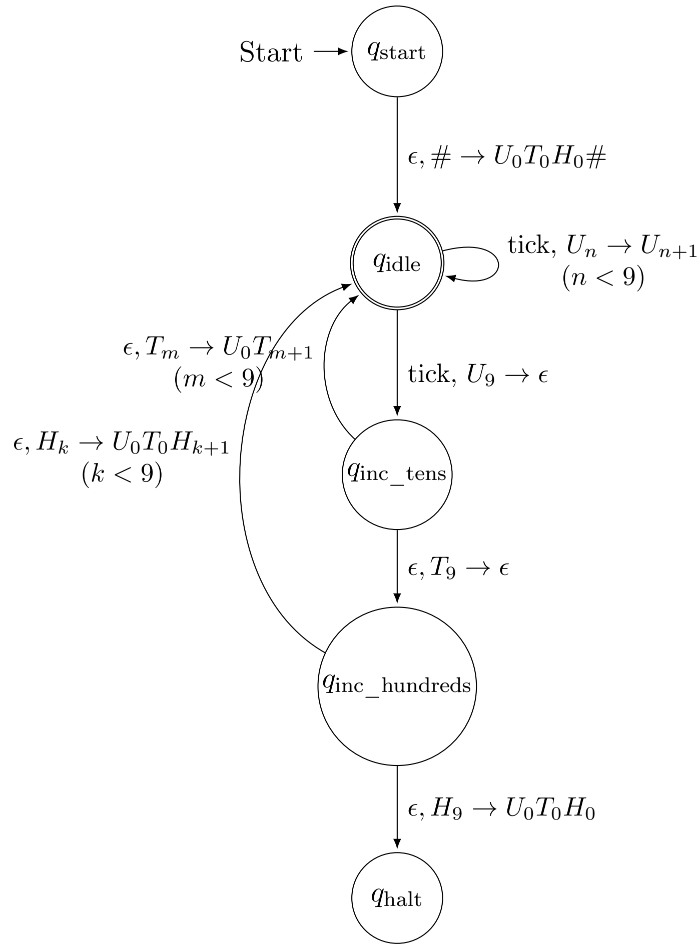

Figure 1: State Diagram for the Three-Digit (0-999) Counter PDA

6 Detailed Example Execution: Counting from 98 to 101

This section demonstrates handling carries across two place values.

1. **Start at 98:** Assume the PDA is in state q_{idle} with stack $(\#, H_0, T_9, U_8)$.
2. **Input 99 ('tick'):**
   - In q_{idle}, reads 'tick'. Top is U_8. Pops U_8, pushes U_9. Stays q_{idle}.
   - Stack: $(\#, H_0, T_9, U_9)$ (represents 99).
3. **Input 100 ('tick'):**
   - In q_{idle}, reads 'tick'. Top is U_9. Pops U_9, pushes nothing. Enters $q_{\text{inc\_tens}}$.
   - Stack: $(\#, H_0, T_9)$.
   - Epsilon transition from $q_{\text{inc\_tens}}$. Top is T_9. Pops T_9, pushes nothing. Enters $q_{\text{inc\_hundreds}}$.
   - Stack: $(\#, H_0)$.
   - Epsilon transition from $q_{\text{inc\_hundreds}}$. Top is H_0. Pops H_0. Pushes H_1, then T_0, then U_0. Enters q_{idle}.

- Stack: $(\#, H_1, T_0, U_0)$ (represents 100).

4. **Input 101 ('tick'):**
   - In q_{idle}, reads 'tick'. Top is U_0. Pops U_0, pushes U_1. Stays q_{idle}.
   - Stack: $(\#, H_1, T_0, U_1)$ (represents 101).

7 Handling Multi-Level Carries

This PDA manages carries across multiple digits using intermediate states:

1. **Units Carry:** U_9 rollover triggers a transition to $q_{\text{inc\_tens}}$, popping U_9 and exposing the tens digit T_Y.
2. **Tens Processing:** $q_{\text{inc\_tens}}$ handles T_Y via epsilon transition.
   - If $Y < 9$, it increments T_Y to T_{Y+1}, pushes the U_0, and returns control to q_{idle}.
   - If $Y = 9$, it pops T_9 and transitions control to $q_{\text{inc\_hundreds}}$, exposing the hundreds digit H_X.
3. **Hundreds Processing:** $q_{\text{inc\_hundreds}}$ handles H_X via epsilon transition.
   - If $X < 9$, it increments H_X to H_{X+1}, pushes T_0, pushes U_0, and returns control to q_{idle}.
   - If $X = 9$, it pushes H_0, T_0, U_0 (representing the rollover part) and transitions to q_{halt} to signal overflow.

This chained state transition correctly simulates the ripple effect of a carry across units, tens, and hundreds places.

8 Formal Transition Function (δ) for Three-Digit Counter

The transition function $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to Q \times \Gamma^*$ is defined as: *(Note: Stack push $(S_1, S_2, \dots)$ pushes S_n first, ..., S_2, then S_1 last/top)*

- **Initialization:**
$$\delta(q_{\text{start}}, \epsilon, \#) = (q_{\text{idle}}, (U_0, T_0, H_0, \#))$$
- **Idle State (Units):** For $n \in \{0, \dots, 8\}$:
$$\delta(q_{\text{idle}}, \text{tick}, U_n) = (q_{\text{idle}}, (U_{n+1}))$$
$$\delta(q_{\text{idle}}, \text{tick}, U_9) = (q_{\text{inc\_tens}}, ())$$
- **Tens Carry State (Epsilon):** For $m \in \{0, \dots, 8\}$:
$$\delta(q_{\text{inc\_tens}}, \epsilon, T_m) = (q_{\text{idle}}, (U_0, T_{m+1}))$$
$$\delta(q_{\text{inc\_tens}}, \epsilon, T_9) = (q_{\text{inc\_hundreds}}, ())$$
- **Hundreds Carry State (Epsilon):** For $k \in \{0, \dots, 8\}$:
$$\delta(q_{\text{inc\_hundreds}}, \epsilon, H_k) = (q_{\text{idle}}, (U_0, T_0, H_{k+1}))$$
$$\delta(q_{\text{inc\_hundreds}}, \epsilon, H_9) = (q_{\text{halt}}, (U_0, T_0, H_0))$$
- **Halt State:**
$$\delta(q_{\text{halt}}, \cdot, \cdot) = \emptyset$$

9 Example: Counting from 998 to Overflow

Tracing the behavior at the upper limit:

1. **Start at 998:** State q_{idle}, Stack $(\#, H_9, T_9, U_8)$.
2. **Input 999 ('tick'):**
   - $\delta(q_{\text{idle}}, \text{tick}, U_8) = (q_{\text{idle}}, (U_9))$.
   - Stack: $(\#, H_9, T_9, U_9)$ (represents 999). State q_{idle}.
3. **Input 1000 ('tick'):**
   - $\delta(q_{\text{idle}}, \text{tick}, U_9) = (q_{\text{inc\_tens}}, ())$.
   - Stack: $(\#, H_9, T_9)$. State $q_{\text{inc\_tens}}$.
   - ϵ-trans: $\delta(q_{\text{inc\_tens}}, \epsilon, T_9) = (q_{\text{inc\_hundreds}}, ())$.
   - Stack: $(\#, H_9)$. State $q_{\text{inc\_hundreds}}$.
   - ϵ-trans: $\delta(q_{\text{inc\_hundreds}}, \epsilon, H_9) = (q_{\text{halt}}, (U_0, T_0, H_0))$.
   - Stack: $(\#, H_0, T_0, U_0)$. State q_{halt}.
4. **State q_{halt}:** The PDA enters the non-accepting halt state. The stack represents '000', but the state signals the overflow. Further 'tick' inputs are rejected as no transitions are defined from q_{halt}.

10 Practical Considerations and Limitations

This three-digit counter PDA successfully models counting within its defined range (0-999).

- **Bounded Range:** The automaton is explicitly designed for three digits.
- **Scalability Limitation (State-Based):** Extending this specific design to, say, ten digits would require ten distinct place-value symbol sets $(U, T, H, Th, \dots)$ and ten corresponding intermediate carry states $(q_{\text{inc\_tens}}, q_{\text{inc\_hundreds}}, \dots)$. While possible, the number of states and transitions grows linearly with the number of digits, making the explicit definition cumbersome for very large, fixed ranges.
- **Contrast with Unbounded Models:** This successful bounded model highlights why the unbounded recursive counter using only simple digit symbols $(D_0..D_9)$ and few states failed with standard PDAs. Managing the carry requires either distinct symbols/stack structure per position or distinct states per carry level, which becomes infinite in the unbounded case unless a more powerful model (like a Turing Machine) is used.
- **Output Interpretation:** Reading the final count involves interpreting the stack configuration $(\#, H_X, T_Y, U_Z)$.

11 Conclusion

By extending the logic used for the two-digit counter, we have designed a formally correct Deterministic Pushdown Automaton capable of counting from 0 to 999. This PDA uses distinct stack symbols for each place value (Units, Tens, Hundreds) and employs intermediate states ($q_{\text{inc\_tens}}$, $q_{\text{inc\_hundreds}}$) to manage the propagation of carries across digit boundaries via epsilon transitions. The model correctly handles multi-digit increments and explicitly halts in a non-accepting state upon overflow, demonstrating that standard PDAs can effectively model counting within a fixed, multi-digit range.

This contrasts with attempts to model unbounded counting using simpler stack representations, confirming that the specific way carries interact with place value requires careful state or stack management that becomes infinite in the unbounded case for standard PDAs. This exercise validates the suitability of PDAs for such bounded counting tasks while illustrating the design patterns needed to handle multi-level dependencies using finite state control and stack manipulation.

11.1 Key Takeaways

- **Fixed-Range Counting with PDAs:** Standard DPDAs can correctly model multi-digit base-10 counting within a predefined range (e.g., 0-999).
- **Hierarchical Carry Management:** Multi-level carries can be managed using a chain of intermediate states, each responsible for processing the carry at a specific place value via epsilon transitions.
- **Stack Representation:** Using distinct symbols for each place value (e.g., H_k, T_m, U_n) is crucial for the state logic to correctly identify and process the appropriate digit during carries.
- **Scalability vs. Boundedness:** While this state-based approach works, its complexity grows with the number of digits, making it practical for bounded ranges but unsuitable for theoretically unbounded counting, which is better modeled by Turing Machines or alternative formalisms.

Python Test Script (0-999)

```python
# Import necessary classes from automata-lib
try:
    from automata.pda.dpda import DPDA
    from automata.pda.stack import PDAStack
    from automata.base.exceptions import RejectionException
except ImportError:
    print("Error: automata-lib not found.")
    print("Please install it: pip install automata-lib")
    # Mocking classes if needed
    class MockPDAConfiguration:
        def __init__(self, state, stack_tuple): self.state, self.stack = state, self.
    _MockStack(stack_tuple)
        class _MockStack:
            def __init__(self, stack_tuple): self.stack = stack_tuple
    class MockDPDA:
        def __init__(self, *args, **kwargs): self.final_states = kwargs.get('final_states
    ', set()); print("Warning: Using Mock DPDA class.")
```

```
        def read_input(self, input_sequence):
             n = len(input_sequence)
             if n > 999: return MockPDAConfiguration('q_halt', ('#', 'H0', 'T0', 'U0'))
             if n == 0: return MockPDAConfiguration('q_idle', ('#', 'H0', 'T0', 'U0'))
             hundreds, rem = divmod(n, 100)
             tens, units = divmod(rem, 10)
             stack_list = ('#', f'H{hundreds}', f'T{tens}', f'U{units}')
             return MockPDAConfiguration('q_idle', tuple(stack_list))
    DPDA = MockDPDA
    RejectionException = Exception
    print("--- automata-lib not found, using Mock classes ---")

import sys

# --- Define the 0-999 Counter PDA ---

# States
states = {'q_start', 'q_idle', 'q_inc_tens', 'q_inc_hundreds', 'q_halt'}

# Input Alphabet
input_symbols = {'tick'}

# Stack Alphabet
stack_symbols = {'#'} | {f'H{i}' for i in range(10)} | \
                        {f'T{i}' for i in range(10)} | \
                        {f'U{i}' for i in range(10)}

# Transitions (Following the successful pattern)
# Remember: Push sequence (S1, S2, S3) pushes S3 first, S2 second, S1 last (top)
transitions = {
    'q_start': {
        '': {
            # Initial: Push #, H0, T0, U0. Stack (#, H0, T0, U0). Top U0.
            '#': ('q_idle', ('U0', 'T0', 'H0', '#'))
        }
    },
    'q_idle': { # Processing Units (top)
        'tick': {
            # Inc Units < 9: Pop Un, Push U(n+1). Stay q_idle.
            **{f'U{n}': ('q_idle', (f'U{n+1}',)) for n in range(9)},
            # Inc Units = 9: Pop U9, Push nothing. Go to q_inc_tens (Tens digit now top).
            'U9': ('q_inc_tens', ())
        }
    },
    'q_inc_tens': { # Epsilon transitions, processing Tens (top)
        '': {
             # Tens digit Tm (m<9): Pop Tm. Push T(m+1), Push U0. Go q_idle.
             **{f'T{m}': ('q_idle', ('U0', f'T{m+1}')) for m in range(9)},
             # Tens digit T9: Pop T9. Push nothing. Go to q_inc_hundreds (Hundreds digit now top).
             'T9': ('q_inc_hundreds', ())
        }
    },
    'q_inc_hundreds': { # Epsilon transitions, processing Hundreds (top)
```

```
        '': {
             # Hundreds digit Hk (k<9): Pop Hk. Push H(k+1), Push T0, Push U0. Go q_idle.
             **{f'H{k}': ('q_idle', ('U0', 'T0', f'H{k+1}')) for k in range(9)},
             # Hundreds digit H9 (Overflow): Pop H9. Push H0, Push T0, Push U0. Go q_halt.
             'H9': ('q_halt', ('U0', 'T0', 'H0'))
        }
    },
    'q_halt': {
        # No transitions out. Any 'tick' input leads to implicit rejection.
    }
}

# Initial state
initial_state = 'q_start'
initial_stack_symbol = '#'
# Final states (only q_idle represents a valid 0-999 count)
final_states = {'q_idle'}

# Create the DPDA instance
try:
    pda = DPDA(
        states=states,
        input_symbols=input_symbols,
        stack_symbols=stack_symbols,
        transitions=transitions,
        initial_state=initial_state,
        initial_stack_symbol=initial_stack_symbol,
        final_states=final_states,
        acceptance_mode='final_state'
    )
    print("DPDA for 0-999 created successfully.")
except Exception as e:
     print(f"Error creating DPDA: {e}")
     # Mock DPDA fallback
     class MockPDAConfiguration:
        def __init__(self, state, stack_tuple): self.state, self.stack = state, self._MockStack(stack_tuple)
        class _MockStack:
             def __init__(self, stack_tuple): self.stack = stack_tuple
     class MockDPDA:
        def __init__(self, *args, **kwargs): self.final_states = kwargs.get('final_states', set()); print("Warning: Using Mock DPDA class after creation error.")
        def read_input(self, input_sequence):
             n = len(input_sequence)
             if n > 999: return MockPDAConfiguration('q_halt', ('#', 'H0', 'T0', 'U0'))
             if n == 0: return MockPDAConfiguration('q_idle', ('#', 'H0', 'T0', 'U0'))
             hundreds, rem = divmod(n, 100); tens, units = divmod(rem, 10)
             stack_list = ('#', f'H{hundreds}', f'T{tens}', f'U{units}')
             return MockPDAConfiguration('q_idle', tuple(stack_list))
     pda = MockDPDA(final_states=final_states)
     RejectionException = Exception
     print("--- Proceeding with Mock PDA ---")

```

```python

# Function to convert the 3-digit stack contents to an integer
def stack_to_int_3digit(stack_tuple: tuple) -> int:
    """
    Converts the PDA stack tuple ('#', HX, TY, UZ) to the integer XYZ.
    """
    # Basic validation
    if not (isinstance(stack_tuple, tuple) and len(stack_tuple) == 4 and \
            stack_tuple[0] == '#' and stack_tuple[1].startswith('H') and \
            stack_tuple[2].startswith('T') and stack_tuple[3].startswith('U')):
        # Allow for initial state stack ('#', 'H0', 'T0', 'U0') during halt
        if not (len(stack_tuple) == 4 and stack_tuple[1:] == ('H0', 'T0', 'U0')):
             print(f"Warning: Invalid stack state for 3-digit conversion: {stack_tuple}")
             return -1

    try:
        # Extract digits, handling potential errors if symbols are wrong
        h_digit = int(stack_tuple[1][1:])
        t_digit = int(stack_tuple[2][1:])
        u_digit = int(stack_tuple[3][1:])
        return h_digit * 100 + t_digit * 10 + u_digit
    except (ValueError, IndexError):
        print(f"Error converting stack digits to int: {stack_tuple}")
        return -2

# --- Testing the PDA ---
print("\nTesting 3-Digit (0-999) Counter PDA:")
# Test cases around boundaries
test_counts = [0, 1, 9, 10, 11, 99, 100, 101, 998, 999, 1000, 1001]

for count in test_counts:
    print(f"\n--- Testing count = {count} ---")
    input_sequence = ['tick'] * count
    try:
        final_config = pda.read_input(input_sequence)
        final_state = final_config.state
        if hasattr(final_config, 'stack') and hasattr(final_config.stack, 'stack'):
             final_stack_tuple = final_config.stack.stack
        else:
             print("Error: Final configuration object has unexpected structure.")
             final_stack_tuple = ('#', 'ERROR', 'ERROR', 'ERROR')

        is_accepted = final_state in pda.final_states # Check if ended in q_idle

        print(f"Input: {count} 'tick's")
        print(f"Ended in State: {final_state}")
        print(f"Final Stack: {final_stack_tuple}")

        expected_acceptance = (count <= 999)

        print(f"Expected Acceptance: {expected_acceptance}")
        print(f"Actual Acceptance: {is_accepted}")

        if is_accepted:
```

```python
            calculated_value = stack_to_int_3digit(final_stack_tuple)
            print(f"Expected Value (if accepted): {count}")
            print(f"Calculated Value: {calculated_value}")
            if calculated_value == count and expected_acceptance:
                print("Result: Correct")
            else:
                print("Result: INCORRECT (Value mismatch or unexpected acceptance)")
        else: # Rejected (ended in q_halt)
            print("Expected Value (if accepted): N/A")
            print("Calculated Value: N/A (Rejected)")
            # Check if rejection was expected (count >= 1000)
            if not expected_acceptance:
                 print("Result: Correct (Rejected as expected)")
            else: # Should not happen for count <= 999
                 print("Result: INCORRECT (Unexpected rejection)")

    except RejectionException as re:
        # This means the PDA got genuinely stuck (no transition defined)
        # Should only happen if input contains something other than 'tick' or logic error
        print(f"Input: {count} 'tick's")
        print(f"PDA Rejection Exception: {re}")
        # Check if this was the expected halt state after 1000+ ticks
        is_halt_state = False
        try:
            # Try reading again to see the state (might not work if truly stuck)
            halt_config = pda.read_input(input_sequence)
            if halt_config.state == 'q_halt':
                is_halt_state = True
        except:
            pass # Ignore errors trying to re-read if stuck

        if not expected_acceptance and is_halt_state:
             print("Result: Correct (Rejected via halt state as expected)")
        else:
             print("Result: REJECTED (Stuck) - Check Logic")

    except Exception as e:
        print(f"Input: {count} 'tick's")
        print(f"PDA Error: {e}")
        # import traceback
        # traceback.print_exc()
        print("Result: ERROR")
```
Listing 1: Python Test Script for 3-Digit PDA (0-999)