Python Introduction Tutorial

November 21, 2023

Code Explanation

Program 1

```
from stack import ListStack

values = ListStack()

for i in range(16):
if i % 3 == 0:
values.push(i)
elif i % 4 == 0:
values.pop()

print(values)
```

Class Definition (`ListStack`):

- **Purpose:** This class defines a simple stack data structure.
- **Methods:**
  - `push`: Adds an item to the top of the stack.
  - `pop`: Removes and returns the top item from the stack. If the stack is empty, it returns `None`.

Main Code Execution:

- A `ListStack` object, `values`, is created.
- A `for` loop iterates from 0 to 15 (inclusive).
- During each iteration:
  - If `i` is a multiple of 3 (`i % 3 == 0`), `i` is added to the stack.
  - If `i` is a multiple of 4 (`i % 4 == 0`) and not a multiple of 3, the top item of the stack is removed.
- The state of the stack is printed at the start of each loop iteration and after the loop concludes.

Loop Iterations and Stack State

- Start of loop (initial state): `[]`
- `i = 0`: `[0]` (0 is pushed)
- `i = 1`: `[0]`
- `i = 2`: `[0]`
- `i = 3`: `[0, 3]` (3 is pushed)
- `i = 4`: `[0]` (3 is popped)
- `i = 5`: `[0]`
- `i = 6`: `[0, 6]` (6 is pushed)
- `i = 7`: `[0, 6]`
- `i = 8`: `[0]` (6 is popped)
- `i = 9`: `[0, 9]` (9 is pushed)
- `i = 10`: `[0, 9]`
- `i = 11`: `[0, 9]`
- `i = 12`: `[0, 9, 12]` (12 is pushed)
- `i = 13`: `[0, 9, 12]`
- `i = 14`: `[0, 9, 12]`
- `i = 15`: `[0, 9, 12]` (Since `i` is a multiple of 3, 15 is pushed but it's not reflected in the loop's output)

Final State of the Stack

Final state of the stack after the loop: `[0, 9, 12, 15]`

Program 2 Explanation and Output

Program 2

```python
from stack import ListStack

def square(n):
        SquareStack=ListStack()
        i=0
        for j in range(1,n,2):
                for k in range(j):
                        SquareStack.push(i)
                        i+=1
                for k in range(j-1):
                        if not SquareStack.is_empty():
                                SquareStack.pop()
                        while SquareStack.top()>n:
                                SquareStack.pop()
        return SquareStack

print(square(100))
```

The `square` Function:

This function takes an integer `n` as input and performs operations on a stack (`ListStack`). It aims to manipulate the stack in a specific pattern and return the modified stack.

- **Initialization:** A `ListStack` named `SquareStack` is created, and a variable `i` is set to 0.
- **Building the Stack:**
  - The outer loop iterates through odd numbers from 1 up to `n`, with a step of 2.
  - For each odd number `j`:
    - The first inner loop pushes incrementing values of `i` onto the stack `j` times, incrementing `i` each time.
    - The second inner loop pops elements from the stack `j-1` times.
  - This pattern creates a sequence of square numbers.
- **Final Adjustment:**
  - A while loop ensures that no number greater than `n` remains in the stack.

Examples

- **For j = 1**: The stack has `0` pushed (1 time), no pop occurs. (`i` becomes 1).
- **For j = 3**: The stack has `1, 2, 3` pushed, then `2, 3` popped. Only `1` remains. (`i` becomes 4).
- **For j = 5**: The stack has `4, 5, 6, 7, 8` pushed, then `5, 6, 7, 8` popped. Only `4` remains. (`i` becomes 9).
- **For j = 7**: The stack has `9, 10, 11, 12, 13, 14, 15` pushed, then `10, 11, 12, 13, 14, 15` popped. Only `9` remains. (`i` becomes 16).
- **For j = 9**: The stack has `16, 17, 18, 19, 20, 21, 22, 23, 24` pushed, then `17, 18, 19, 20, 21, 22, 23, 24` popped. Only `16` remains. (`i` becomes 25).
- **For j = 11**: The stack has `25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35` pushed, then `26, 27, 28, 29, 30, 31, 32, 33, 34, 35` popped. Only `25` remains. (`i` becomes 36).

Output:

When the function is called with `n=100`, the output is a stack containing the elements:

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

This output reveals that the function generates and stores square numbers up to and including `n`, provided `n` itself is a square number.

Translation of Infix to Postfix

i) Infix to Postfix:

1. $V \times W \times (X + Y) - Z$

   Derivation:

$$\begin{aligned}
&\text{Step 1: } X + Y \rightarrow XY+ \\
&\text{Step 2: } VW \times XY+ \rightarrow VW \times XY+* \\
&\text{Step 3: } VW \times XY+* - Z \rightarrow VW \times XY+*Z- \\
\textbf{Postfix: } & VW * XY + *Z-
\end{aligned}$$

2. $X - (Y + W \times (Z/V))$

   Derivation:

$$\begin{aligned}
&\text{Step 1: } Z/V \rightarrow ZV/ \\
&\text{Step 2: } W \times ZV/ \rightarrow WZV/* \\
&\text{Step 3: } Y + WZV/* \rightarrow YWZV/*+ \\
&\text{Step 4: } X - YWZV/*+ \rightarrow XYWZV/*+- \\
\textbf{Postfix: } & XYWZV/*+-
\end{aligned}$$

3. $A - B \times (C + D/E)$

   Derivation:

$$\begin{aligned}
&\text{Step 1: } D/E \to DE/ \\
&\text{Step 2: } C + DE/ \to CDE/+ \\
&\text{Step 3: } B \times CDE/+ \to BCDE/+* \\
&\text{Step 4: } A - BCDE/*+ \to ABCDE/+*- \\
\textbf{Postfix: } &ABCDE/+*-
\end{aligned}$$

4. $A/(B \times C) - (D + E)$

   Derivation:

$$\begin{aligned}
&\text{Step 1: } B \times C \to BC* \\
&\text{Step 2: } A/BC* \to ABC*/ \\
&\text{Step 3: } D + E \to DE+ \\
&\text{Step 4: } ABC/*-DE+ \to ABC/*DE+- \\
\textbf{Postfix: } &ABC*/DE+-
\end{aligned}$$

Translation of Postfix to Infix

ii) Postfix to Infix:

Expression (a): $ABC - D * +$

1. **Postfix:** $ABC - D * +$
2. **Steps:**
   - Start from left, AB (operands), and C (operand), followed by an operator $-$. In infix, this becomes $(B - C)$.
   - Now, we have $D(B - C)$. The next operator is $*$, which applies to D and $(B - C)$, making it $D * (B - C)$.
   - Finally, the last operator $+$ applies to the whole expression and another operand D, resulting in $A + (B - C) * D$.
   - Rearranging for clarity: $(A + ((B - C) * D))$.
3. **Infix:** $(A + ((B - C) * D))$

Expression (b): $XYZ + AB - *-$

1. **Postfix:** $XYZ + AB - *-$
2. **Steps:**
   - Start with XY, then Z, followed by $+$, translating to $(Y + Z)$.
   - Next, we encounter AB, and $-$, translating to $(A - B)$.

- The $*$ operator applies to $(Y+Z)$ and $(A-B)$, giving $(Y+Z)*(A-B)$.
- Finally, the $-$ operator is applied between X and the entire $(Y+Z)*(A-B)$ expression, resulting in $X-((Y+Z)*(A-B))$.

3. **Infix:** $X-((Y+Z)*(A-B))$

Expression (c): $AB+CD-/E+$

1. **Postfix:** $AB+CD-/E+$
2. **Steps:**
   - Starting with $AB+$, this becomes $(A+B)$.
   - Moving on, we have $CD-$, translating to $(C-D)$.
   - The / operator then applies to $(A+B)$ and $(C-D)$, resulting in $((A+B)/(C-D))$.
   - Finally, the + operator combines this result with E, yielding $(((A+B)/(C-D))+E)$.
3. **Infix:** $(((A+B)/(C-D))+E)$

Expression (d): $AB+C-DE*+$

1. **Postfix:** $AB+C-DE*+$
2. **Steps:**
   - Start with $AB+$, converting to $(A+B)$.
   - Next, we see C, and then a $-$ operator, which applies to $(A+B)$ and C, giving $((A+B)-C)$.
   - Following this, $DE*$ translates to $(D*E)$.
   - The final + operator combines $((A+B)-C)$ and $(D*E)$, resulting in $(((A+B)-C)+(D*E))$.
3. **Infix:** $(((A+B)-C)+(D*E))$

Code Execution and Output

Initial Configuration

- **Initial Stack** S: [3, 6, 78, 9] (Top $\rightarrow$ 9)
- **Initial Queue** Q: []

Execution Steps

1. **First Call to CheckElement:**
   - Pops 9 from S: $S = [3, 6, 78]$, $Q = [9]$
   - Since $9 \neq 78$, calls CheckElement again.
2. **Second Call to CheckElement:**
   - Pops 78 from S: $S = [3, 6]$, $Q = [9, 78]$
   - Since $78 = 78$, prints "78 is an element of S."
3. **Restoring the Stack**:
   - Move all elements from Q back to S.
   - $S = [3, 6, 9, 78]$, $Q = []$, but the last two numbers order in S is reversed.
   - Move elements from S to Q and back to S to restore original order.
   - Final $S = [3, 6, 78, 9]$ (original order).

Key Points

- Stack S holds the elements, queue Q is used for temporary storage to maintain order.
- Function uses recursion to check each element in S.
- Double transfer between S and Q restores original order of elements in S.

Output

```
before: [3, 6, 78, 9]
78 is an element of S.
after: [3, 6, 78, 9]
```

1 Introduction

This document outlines the design and implementation of a discrete event simulation for a single-server queuing system, known as the M/M/1 queue. The simulation aims to analyze customer waiting times and queue length in a system where customer arrivals and service times follow an exponential distribution.

M/M/1 Queue

The M/M/1 queue is a fundamental model in queueing theory, which is the study of waiting lines. It is defined by the following characteristics:

1. **M (First M):** Stands for "Markovian" arrival process. Arrivals occur according to a Poisson process, with times between arrivals being exponentially distributed.
2. **M (Second M):** Indicates a "Markovian" service process. The service times are also exponentially distributed.
3. **1:** Represents the number of servers in the system, which is one in this case.

Key Characteristics

- **Single Server:** Only one server is available for serving customers.
- **Exponential Inter-Arrival and Service Times:** Both arrivals and services follow an exponential distribution.
- **First-Come, First-Served:** The service discipline is usually assumed to be first-come, first-served.
- **Infinite Queue Capacity:** The queue can accommodate an unlimited number of customers.
- **Steady-State Analysis:** The system is often analyzed in a steady state, where its properties do not change over time.

Performance Metrics

- **Average Number of Customers in the System (L):** Includes both customers being served and those waiting in the queue.
- **Average Number of Customers in the Queue (Lq):** The average number of customers waiting for service.
- **Average Time a Customer Spends in the System (W):** Total time spent by a customer in the system.
- **Average Time a Customer Spends in Queue (Wq):** Average waiting time for a customer before their service starts.
- **Server Utilization Factor (ρ):** The fraction of time the server is busy.

2 Simulation Objectives

1. Calculate the average waiting time for each customer.
2. Determine the queue length at the end of the simulation period.

3 Simulation Parameters

- Total simulation duration in discrete time units.
- Customer arrival rate (λ), following an exponential distribution.
- Service rate (μ), also following an exponential distribution.

4 Implementation Details

4.1 Classes and Methods

- **Customer Class:** Private data fields for customer ID and arrival time.
- **Server Class:** Methods for starting and stopping service.
- **Simulation Class:** Runs the simulation, handles events like customer arrivals, service initiation, and service completion.

4.2 Simulation Process

1. Initialize with $\lambda = 9$, $\mu = 10$, and total time = 480 minutes.
2. Run the simulation in a loop, processing events at each time step.
3. Generate customers, place them in a queue, and have the server process them.
4. Calculate the total waiting time and queue length at the end.

5 Results and Analysis

- Run the simulation multiple times (1000 times) to average the results.
- Compare the empirical results with theoretical values:
  - Average queue length $L = \frac{\rho}{1-\rho}$ where $\rho = \frac{\lambda}{\mu}$.
  - Average waiting time $T = \frac{\rho}{\mu-\lambda}$.

6 Conclusion

This simulation provides insights into the behavior of queues under various arrival and service rates, which is crucial for numerous real-world applications.