BYU IDAHO

CSE 212 | Programming with Data Structures

W04 Prepare: Reading

Table of Contents

Queues

During the last lesson, we learned about the stack. The Stack was "Last In, First Out" (LIFO) and was implemented using the Python list. The **queue** is characterized as "First In, First Out" (FIFO) and can also be implemented using the Python list.

I. GROCERY STORY QUEUE

In the example below, we can see a line at a busy grocery store used to represent a queue. The person next in line for the cashier is called the **front** and the person at the end of the line is called the **back**. When the person at the front is removed from the queue we call this a **dequeue** operation. When a new person joins the queue at the back, we call this an **enqueue** operation. Note that someone cannot cheat and enter the line in the middle of the queue.

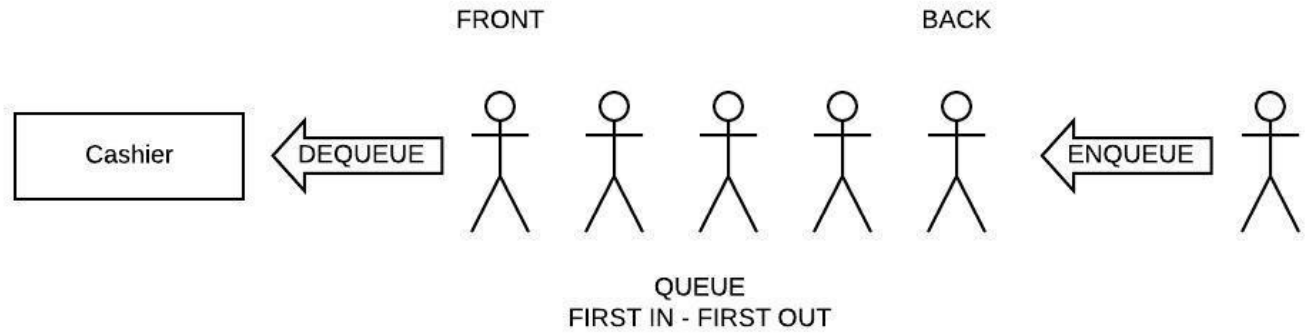

Figure 1 - Grocery Store Line Queue

Queues are used when we need to process a collection of requests in a fair and orderly way. Consider the following two common queues used in software: the Web Server Queue and the Reader/Writer Queue.

II. WEB SERVER QUEUE

A web server receives numerous HTTP (Hypertext Transfer Protocol) requests for web pages from clients throughout the world. Each request requires the web server to send back information. The amount of time it takes to send that information makes it difficult to respond timely to all requests. This would be similair to a customer service desk that had only one phone. If the customer service agent is helping someone else, then no one would pick up your call. To solve the problem, a queue is used to pick up all the phone calls and

transfer you to the customer service agent when they are ready for the next person.

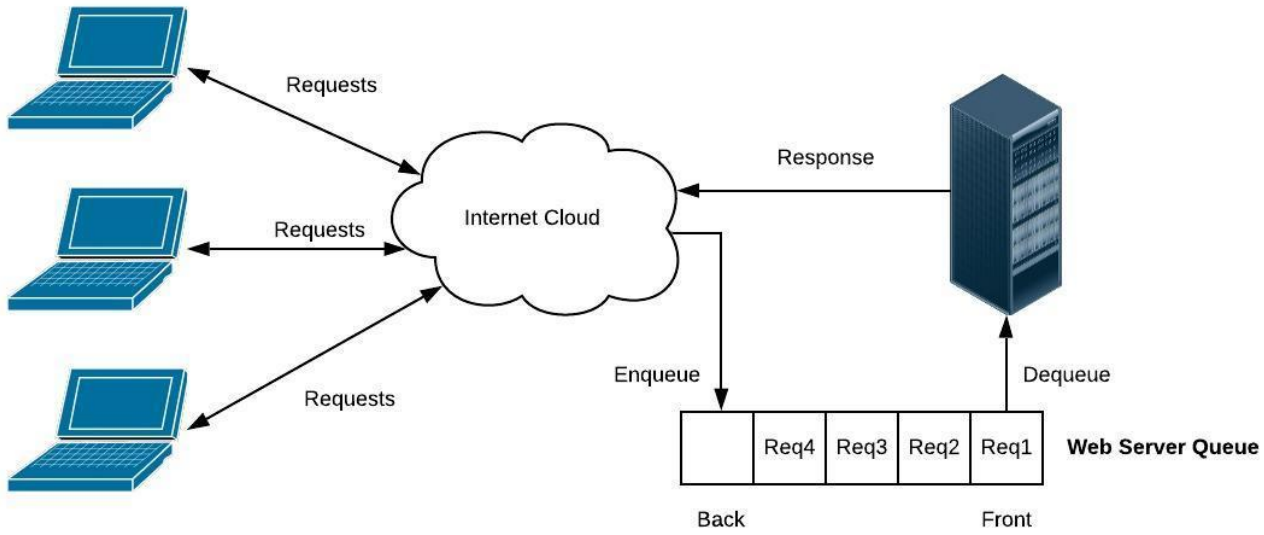

Figure 2 - Web Server Queue

The web server does the same thing. When a request is sent, it is put into a queue until the web server can process the request. In this way, all requests are received and none of them are lost. Queues frequently have a self-imposed maximum size. If a queue is full, then the software may need to send an error message back to the client.

III. READER/WRITER QUEUE

Frequently, we have the need to run different software components concurrently (e.g. looks like they are running at the same time). Each component is called a **process** or a thread (additional information about threads in Python can be found here). Each process will likely have their own set of variables that are maintained. Frequently, there is need to have shared data between the processes. The diagram below shows a variable which is being shared by multiple processes.

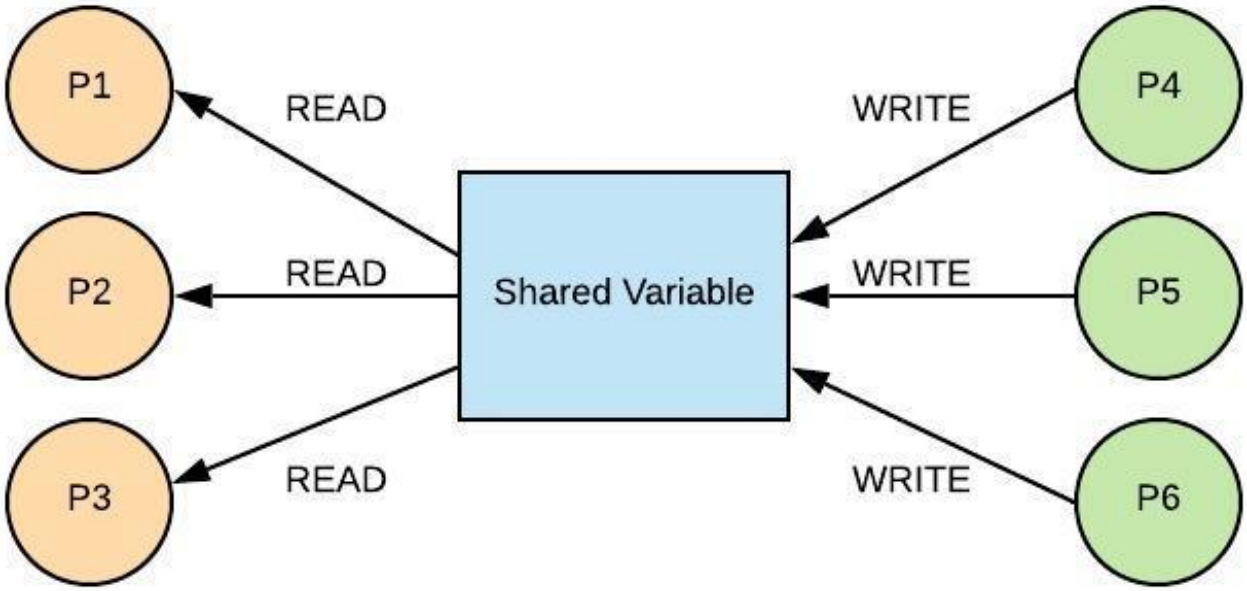

Figure 3 - Reader Writer Problem

Processes P1 through P6 are all trying to use the variable at the same time. Processes P1, P2, and P3 are reading the variable and processes P4, P5, and P6 are writing to the variable. The concurrent reading is not a problem. However, if everyone tries to both read and write at the same time, new and modified values may be missed or overwritten. One solution is to protect the code that is writing to the shared data so that only one process can change the variable at a time. A queue is used to ensure order and integrity. When a process wants to write, it is enqueued. When a process is dequeued, it is then allowed to modify the shared variable. When the process is done, then the next process is dequeued.

IV. QUEUES IN PYTHON

In Python, a queue can be represented using a list. To dequeue an item from the front of the queue, `[0]` and `del` can be used to both obtain and delete the first item from the list. To enqueue an item to the back of the queue, the `append` function can be used on the list. The size can be determined by using the `len` function on the list. The

performance of the queue using a Python list is based on the performance of the dynamic array.

Common Queue Operation	Description	Python Code	Performance
enqueue(value)	Adds "value" to the back of the queue	`my_queue.append(value)`	O(1) - Performance of adding to the end of the dynamic array

dequeue()	Two approaches: Remove and return the item from the front of the queue; or pop off index 0	`value = my_queue[0]`
		`del my_queue[0]`
		or
		`value = my_queue.pop(0)`
			O(n) - Performance of obtaining and removing from the beginning of the dynamic array

size()	Return the size of the queue	`length = len(my_queue)`	O(1) - Performance of returning the size of the dynamic array
empty()	Returns true if the length of the queue is zero.	`if len(my_queue) == 0:`	O(1) - Performance of checking the size of the dynamic array

The Python library also includes a class called **`deque`** which stands for double ended queue and is more frequently used due to better performance. We will learn more about this in the future when we study linked lists (Lesson 7). It is important to note at this point that when we use a Queue with a Linked List instead of Dynamic Array, then we will have better performance with the **`dequeue`** function.

Finding Defects Using Testing

The best time to find a **defect** is while you are "in-phase." This means that before you deliver your software to the customer you want to identify as many errors as possible. It is much more expensive to fix errors if the software has already been delivered to the customer. The two most common methods for finding defects are:

- Code Review
- Testing

I. TESTING CODE FROM REQUIREMENTS

Even the best code reviewers may not be able to analyze all the different scenarios that the software will execute within. **Testing** is a process of demonstrating that specific inputs will result in expected outputs. The selection of inputs can be done systematically. We do not need to test all input combination. Often times we will write code to test your code. Consider a program that was supposed to determine if a year was a leap year. The **requirements** of a program would include the following:

- Every 4 years shall be a leap year.
- Every 100 years shall not be a leap year.
- Every 400 years shall be a leap year.

Based on these requirements, we can write some **test cases**. Notice that we are not writing test cases based on what the code does. Each test case includes an **expected result**. If there is an error in the code, we may incorrectly write the test case based on the faulty code. Using the requirements above, we may write the following tests:

- Test 1

Scenario: Year 1996 (multiple of 4 but not multiple of 100 or 400)
Expected Result: True

- Test 2

  Scenario: Year 1900 (multiple of 4, multiple of 100, not multiple of 400)
  Expected Result: False

- Test 3

  Scenario: Year 2000 (multiple of 4, multiple of 100, multiple of 400)
  Expected Result: True

- Test 4

  Scenario: Year 2003 (not multiple of 4, 100, or 400)
  Expected Result: False

II. RUNNING TEST CASES

Notice that each test has a detailed scenario and expected result based on the requirements. If we were given a function called **`is_leap_year`**, we could write test code as follows:

```
result = is_leap_year(1996)
print(result)
result = is_leap_year(1900)
print(result)
result = is_leap_year(2000)
print(result)
result = is_leap_year(2003)
print(result)
```

If anything fails when we run the test code, then we must look for code related to the test that failed.

Instead of printing out the results, test code in Python can use the **`assert`** function. If the **`assert`** function fails, then the program will exit and tell you which test (e.g. assert statement) failed. For example:

```
assert is_leap_year(1996) == True
assert is_leap_year(1900) == False
```

```
assert is_leap_year(2000) == True
assert is_leap_year(2003) == False
```

For more complicated programs, a single test scenario may require you to call multiple functions to properly set up the scenario. For example, if we were testing the enqueue and dequeue functions of a queue class, we might enqueue three numbers and then dequeue the three numbers to ensure that they came out in the correct order. The test code may look like the following:

```
# Test 1
# Scenario: Ensure that after adding 3 items to the queue, they can be
#           removed in the proper order
# Expected Result: 100, 200, 300
print("Test 1")
queue = Queue()
queue.enqueue(100)
queue.enqueue(200)
queue.enqueue(300)
result = queue.dequeue()
print(result)
result = queue.dequeue()
print(result)
result = queue.dequeue()
print(result)
```

In addition to finding defects, testing also has the benefit of helping the programmer better understand the requirements of the software. Whether you or another engineer wrote the code, the process of writing test scenarios will increase your understanding of what the software should do.

Key Terms

» **back** - Refers to the location in the queue where an enqueue occurs. The last item put in the queue is found in the back.

» **defect** - This is an error in code.

» **dequeue** - The operation to remove an item from the queue. The item is removed from the front of the queue.

» **enqueue** - The operation to add an item to the queue. The item is added to the back of the queue.

» **expected result** - The result that you expect to receive when you run a test case. The expected result is based on your understanding of the software requirements.

» **front** - Refers to the location in the queue where a dequeue occurs. The first item put in the queue is found in the front. Traditionally this is index 0 of a dynamic array.

» **process** - The place where software runs. A process has code that is executed and memory used for variables in that code. Multiple processes can run at the same time. Processes can share memory.

» **queue** - A data structure that follows a First In, First Out (FIFO) rule. The queue is used both to maintain order in data and to remember data when there is not time to process it.

» **requirements** - Written description of what software should do.

» **test cases** - Scenarios that are written to test that code behaves per the requirements. A test case will usually have test code unless the procedure is for the user to interact with the software. An expected result is written for each test case.

» **testing** - An activity to demonstrate that the code correctly implements the requirements.