Chapter 35
ADTs, Queues and Stacks

35.1 Introduction

There are a number of common data structures that are used within computer programs that you might expect to see within Python's list of collection or container classes; these include Queues and Stacks. However, in the basic collection classes these are missing. However, we can either create our own implementations or use one of the extension libraries that provide such collections. In this chapter we will explore implementing our own versions.

35.2 Abstract Data Types

The Queue and Stack are concrete examples of what are known as Abstract Data Types (or ADTs).

An Abstract Data Type (or ADT) is a model for a particular type of data, where a data type is defined by its behaviour (or semantics) from the point of view of the *user* of that data type. This behaviour is typically defined in terms of possible values, possible operations on the data of this type and behaviour of the operations provided by the data type.

An ADT is used to define a common concept that can be implemented by one or more concrete data structures. These implementations may use different internal representations of the data or different algorithms to provide the behaviour; by semantically they meet the descriptions provided by the ADT.

For example, a List ADT may be defined that defines the operations and behaviour that a List like data structure must provide. Concrete implementations may meet the semantics of a List using an underlying array of elements, or by linking elements together with pointers or using some form of hash table (all of which are different internal representations that could be used to implement a list).

J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_35

35.3 Data Structures

We will look at how the Python collection types can be used as both a Queue and a Stack but first we need to define both these ADTs:

- **Queue** is an ADT that defines how a collection of entities are managed and maintained. Queues have what is known as a First-In-First-Out (or FIFO) behaviour that is the first entity added to a queue is the first thing removed from the queue. Within the queue the order in which the entities were added is maintained.
- **Stack** is another ADT but this time it has a Last-In-First-Out (or LIFO) behaviour. That is the most recently added entity to the Stack will be the next entity to be removed. Within the stack the order that the entities were added in is maintained.

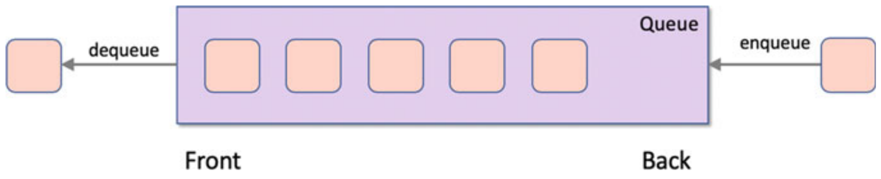
35.4 Queues

Queues are very widely used within Computer Science and in Software Engineering. They allow data to be held for processing purposes where the guarantee is that the earlier elements added will be processed before later ones.

There are numerous variations on the basic queue operations but in essence all queues provide the following features:

- Queue creation.
- Add an element to the back of the queue (known as enqueuing).
- Remove an element from the front of the queue (known as dequeuing).
- Find out the length of the queue.
- Check to see if the queue is empty.
- Queues can be of fixed size or variable (growable) in size.

The basic behaviour of a queue is illustrated by:



In the above diagram there are five elements in the queue, one element has already been removed from the front and another is being added at the back. Note that when one element is removed from the front of the queue all other element move forward one position. Thus, the element that was the second to the front of the queue becomes the front of the queue when the first element is dequeued.

Many queues also allow features such as:

- *Peek* at the element at the front of the queue (that is see what the element is but do not remove it from the queue).
- Provide *priorities* so that elements with a higher priority are not added to the back of the queue but to a point in the middle of the queue related to their priority.

35.4.1 Python List as a Queue

The Python `List` container can be used as a queue using the existing operations such as `append()` and `pop()`, for example:

```
queue = [] # Create an empty queue
queue.append('task1')
print('initial queue:', queue)
queue.append('task2')
queue.append('task3')
print('queue after additions:', queue)
element1 = queue.pop(0)
print('element retrieved from queue:', element1)
print('queue after removal', queue)
```

The output of which is

```
initial queue: ['task1']
queue after additions: ['task1', 'task2', 'task3']
element retrieved from queue: task1
queue after removal ['task2', 'task3']
```

Note that each *task* was added to the end of the queue, but the first task obtained from the queue was task 1.

35.4.2 Defining a Queue Class

In the last section we used the `List` class as a way of providing a queue; this approach does work but it is not obvious that we are using the list as a queue (with the exception of the name of the variable that we are holding the `List` in). For example we have used `pop(0)` to dequeue an element from the queue and we have used `append()` to enqueue an element. In addition there is nothing to stop a programmer forgetting to use `pop(0)` and instead using `pop()` which is an easy mistake to make and which will remove the most recently added item from the queue.

It would be better to create a new data type and ensure that this data provides the queue like behaviour and hide the list inside this data type.

We can do this by defining our own `Queue` class in Python.

```
class Queue:
def __init__(self):
    self._list = [] # initial internal data

def enqueue(self, element):
    self._list.append(element)

def dequeue(self):
    return self._list.pop(0)

def __len__(self):
    """ Supports the len protocol """
    return len(self._list)

def is_empty(self):
    return self.__len__() == 0

def peek(self):
    return self._list[0]

def __str__(self):
    return 'Queue: ' + str(self._list)
```

This `Queue` class internally holds a list. Note we are using the convention that the internal list instance variable name is preceded by an underbar (`'_'`) thereby indicating that no one should access it directly.

We have also defined methods for dequeuing and enqueuing elements to the queue. To complete the definition, we have also defined methods for checking the current length of the queue, whether the queue is empty or not, allowing the element at the front of the queue to be peeked at and of course proving a string version of the queue for printing.

Note that the `is_empty()` method uses the `__len__()` method when determining if the queue is empty; this is an example of an important idea; only define something once. As we want to use the length of the queue to help determine if the queue is empty, we reuse the `__len__()` method rather than the code implementing the length method; thus, if the internal representation changes we will not affect the `is_empty()` method.

The following short program illustrate show the `Queue` class can be used:

```
queue = Queue()
print('queue.is_empty():', queue.is_empty())
queue.enqueue('task1')
print('len(queue):', len(queue))
queue.enqueue('task2')
queue.enqueue('task3')
print('queue:', queue)
print('queue.peek():', queue.peek())
print('queue.dequeue():', queue.dequeue())
print('queue:', queue)
```

The output from this is:

```
initial queue: ['task1']
queue after additions: ['task1', 'task2', 'task3']
element retrieved from queue: task1
queue after removal ['task2', 'task3']
queue.is_empty(): True
len(queue): 1
queue: Queue: ['task1', 'task2', 'task3']
queue.peek(): task1
queue.dequeue(): task1
queue: Queue: ['task2', 'task3']
```

This provides a far more explicit and semantically more meaningful implementation of a `Queue` than the use of the raw `List` data structure.

Of course, Python understands this and provides a queue container class in the `collections` module called `deque`. This implementation is optimised to be more efficient than the basic `List` which is not very efficient when it comes to popping elements from the front of the list.

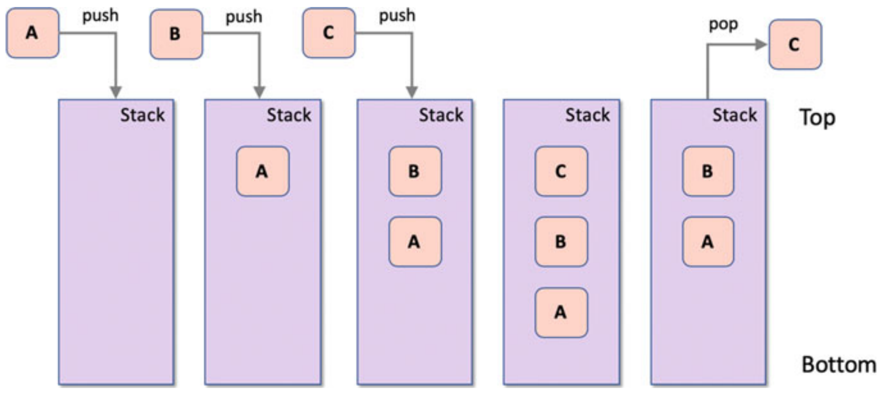
35.5 Stacks

Stacks are another very widely used ADT within computer science and in software applications. They are often used for evaluating mathematical expressions, parsing syntax, for managing intermediate results etc.

The basic facilities provided by a Stack include:

- Stack creation.
- Add an element to the top of the stack (known as pushing onto the stack).
- Remove an element from the top of the stack (known as popping from the stack).
- Find out the length of the stack.
- Check to see if the stack is empty.
- Stacks can be of fixed size or a variable (growable) stack.

The basic behaviour of a stack is illustrated by:



This diagram illustrates the behaviour of a Stack. Each time a new element is pushed onto the Stack, it forces any existing elements further down the stack. Thus the most recent element is at the *top* of the stack and the oldest element is at the *bottom* of the stack. To get to older elements you must first pop newer elements off the top etc.

Many stacks also allow features such as

- *Top* which is often an operation that allows you to peek at the element at the top of the stack (that is see what the element is but do not remove it from the queue).

35.5.1 Python List as a Stack

A `List` may initially appear particularly well suited to being used as a `Stack` as the basic `append()` and `pop()` methods can be used to emulate the stack behaviour. Whatever was most recently appended to the list is the element that will be next returned from the `pop()` method, for example:

```
stack = [] # create an empty stack
stack.append('task1')
stack.append('task2')
stack.append('task3')
print('stack:', stack)
top_element = stack.pop()
print('top_element:', top_element)
print('stack:', stack)
```

Which produces the output:

```
stack: ['task1', 'task2', 'task3']
top_element: task3
stack: ['task1', 'task2']
```

This certainly works although when we print out the stack it does not make it clear that 'task3' is at the front of the stack.

In addition, as when using the `List` as a queue ADT it is still possible to apply any of the other methods defined on a `List` to this *stack* and thus we can still write `stack.pop(0)` which would remove the very first element added to the stack.

We could therefore implement a `Stack` class to wrap the list and provide suitable stack like behaviour as we did for the `Queue` class.

35.6 Online Resources

For more information on ADTs, Queues and Stacks see:

- https://en.wikipedia.org/wiki/Abstract_data_type Wikipedia page on ADTs (Abstract Data Types).
- https://en.wikipedia.org/wiki/Queue_(abstract_data_type) Wikipedia page on the Queue data structure.
- https://en.wikipedia.org/wiki/Stack_(abstract_data_type) Wikipedia page on Stacks.
- https://en.wikibooks.org/wiki/Data_Structures/Stacks_and_Queues Wikibooks tutorial on Stack and Queue data structures.

35.7 Exercises

Implement your own `Stack` class following the pattern used for the `Queue` class.

The `Stack` class should provide

- A `push(element)` method used to add an element to the `Stack`.
- A `pop()` method to retrieve the top element of the `Stack` (this method removes that element from the stack).
- A `top()` method that allows you to peek at the top element on the stack (it does not remove the element from the stack).
- A `__len__()` method to return the size of the stack. This method also meets the `len` protocol requirements.
- An `is_empty()` method that checks to see if the stack is empty.
- A `__str__()` method used to convert the stack into a string.

Once completed you should be able to run the following test application:

```
stack = Stack()
stack.push('T1')
stack.push('T2')
stack.push('T3')
print('stack:', stack)
print('stack.is_empty():', stack.is_empty())
print('stack.length():', stack.length())
print('stack.top():', stack.top())
print('stack.pop():', stack.pop())
print('stack:', stack)
```

An example of the type of output this might produce is

```
stack: ['task1', 'task2', 'task3']
top_element: task3
stack: ['task1', 'task2']
stack: Stack: ['T1', 'T2', 'T3']
stack.is_empty(): False
stack.length(): 3
stack.top(): T3
stack.pop(): T3
stack: Stack: ['T1', 'T2']
```