6.2.7 Using Lists as Stack

Stack is an important data structure which stores its elements in an ordered manner. We will explain the concept of stacks using an analogy. You must have seen a pile of plates where one plate is placed on top of another as shown in Figure 6.1. Now, when you want to remove a plate, you remove the topmost plate first. Hence, you can add and remove an element (i.e. a plate) only at/from one position which is the topmost position.

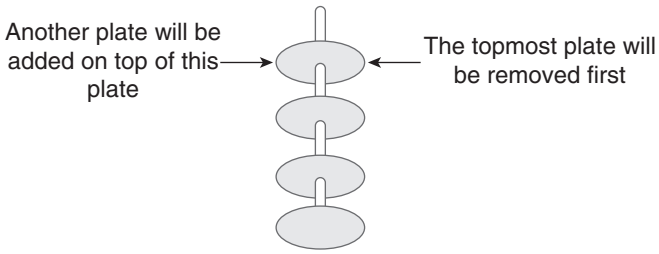

Figure 6.1 A stack of plates

Stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end. Hence, a stack is called a **LIFO** (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.

Now the question is, where do we need stacks in computer science? The answer is in function calls. Consider an example, where we are executing function `A`. In the course of its execution, function `A` calls another function `B`. Function `B` in turn calls another function `C`, which calls function `D`. In order to keep track of the returning point of each active function, a special stack called system stack or call stack is used. Whenever a function calls another function, the calling function is pushed onto the top of the stack. This is because after the called function gets executed, the control is passed back to the calling function. Look at Figure 6.2 which shows this concept.

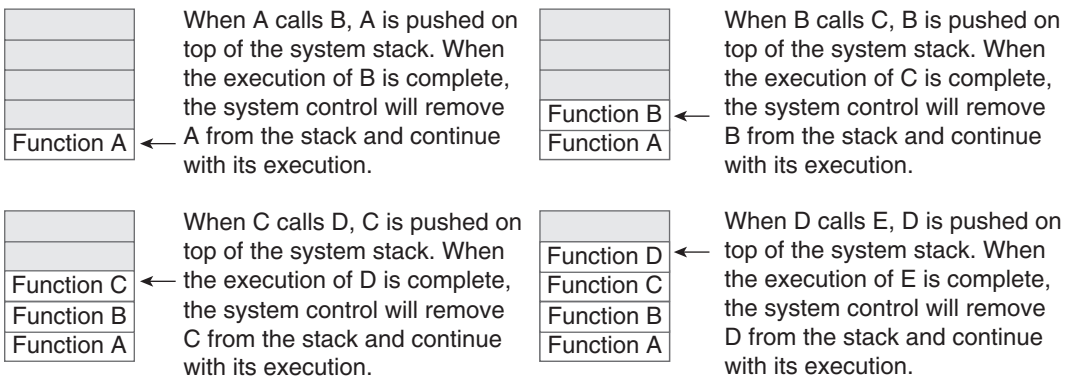

Figure 6.2 Calling function from another function

Now, when function `E` is executed, function `D` will be removed from the top of the stack and executed. Once function `D` gets completely executed, function `C` will be removed from the stack for execution. The whole procedure will be repeated until all the functions get executed. Let us look at the stack after each function is executed. This is shown in Figure 6.3. The system stack ensures a proper execution order of functions. Therefore, stacks are frequently used in situations where the order of processing is very important, especially when the processing needs to be postponed until other conditions are fulfilled.

A stack supports three basic operations: *push, pop*, and *peep (or peek).* The `push` operation adds an element at the end of the stack. The `pop` operation removes the last element from the stack. And, the `peep` operation returns the value of the last element of the stack (without deleting it). In Python, the list methods make it very easy to use a list as a stack. For example, to push an element in the stack, you will use the `append()` method,

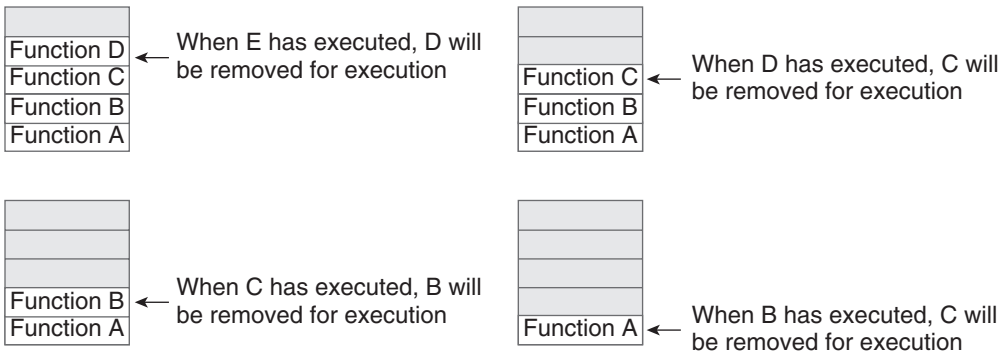

Figure 6.3 Returning from called functions

to pop an element use the `pop()` method, and for peep operation use the slicing operation as illustrated in the program given below.

Example 6.1 Program to illustrate operations on a stack

```
stack = [1,2,3,4,5,6]
print("Original stack is : ", stack)
stack.append(7)
print("Stack after push operation is : ", stack)
stack.pop()
print("Stack after pop operation is : ", stack)
last_element_index = len(stack) - 1
print("Value obtained after peep operation is : ",
stack[last_element_index])
```

OUTPUT

```
Original stack is :  [1, 2, 3, 4, 5, 6]
Stack after push operation is :  [1, 2, 3, 4, 5, 6, 7]
Stack after pop operation is :  [1, 2, 3, 4, 5, 6]
Value obtained after peep operation is :  6
```

Note The `del` statement and the `pop()` method does the same thing. The only difference between them is that `pop()` returns the removed item.

6.2.8 Using Lists as Queues

Queue is an important data structure which stores its elements in an ordered manner. For example, consider the analogies given below.

- People moving on an escalator. The people who got on the escalator first will be the first one to step out of it.
- People waiting for a bus. The first person standing in the line will be the first one to get into the bus.
- People standing outside the ticketing window of a cinema hall. The first person in the line will get the ticket first and thus will be the first one to move out of it.

- Luggage kept on conveyor belts. The bag which was placed first will be the first to come out at the other end.
- Cars lined at a toll bridge. The first car to reach the bridge will be the first to leave.

In all these examples, we see that the element at the first position is served first. Same is the case with queue data structure. A queue is a **FIFO** (First-In-First-Out) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end and removed from the other end. In computer systems, the operating system makes full use of queues for the following tasks.

- To maintain waiting lists for a single shared resource like printer, disk, CPU, etc.
- To transfer data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., pipes, file IO, and sockets.
- As buffers on MP3 players and portable CD players, iPod playlist, etc.
- Handling interrupts. When programming a real-time system that can be interrupted, for example, by a mouse click, it is necessary to process the interrupts immediately; before proceeding with the current job. If the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriate data structure.
- Queues are also used in the playlist of jukebox to add songs to the end and play from the front of the list.

Queue supports three basic operations—*insert, delete,* and *peep (or peek).* In Python, you can easily implement a queue by using the `append()` method to insert an element at the end of the queue, `pop()` method with an index `0` to delete the first element from the queue, and slice operation to print the value of the last the element in the queue. The program given below illustrates this concepts.

Example 6.2 Program to show the implementation of a queue using list data structure

```
queue = [1,2,3,4,5,6]
print("Original queue is : ", queue)
queue.append(7)
print("Queue after insertion is : ", queue)
queue.pop(0)
print("Queue after deletion is : ", queue)
print("Value obtained after peep operation is : ", queue[(len(queue) - 1)])
```

OUTPUT

```
Original queue is :  [1, 2, 3, 4, 5, 6]
Queue after insertion is :  [1, 2, 3, 4, 5, 6, 7]
Queue after deletion is :  [2, 3, 4, 5, 6, 7]
Value obtained after peep operation is :  7
```

6.5 SETS

Sets is another data structure supported by Python. Basically, sets are same as lists but with a difference that sets are lists with no duplicate entries. Technically, a set is a mutable and an unordered collection of items. This means that we can easily add or remove items from it.

6.5.1 Creating a Set

A set is created by placing all the elements inside curly brackets {}, separated by comma or by using the built-in function `set()`. The syntax of creating a set can be given as,

```
set_variable = {val1, val2, ...}
```

For example, to create a set you can write,

```
>>> s = {1,2.0,"abc"}
>>> print( s)
set([1, 2.0,'abc'])
```

A set can have any number of items and they may be of different data types. For example, the code given below creates a set using the `set()` function. The code converts a list of different types of values into a set.

Example 6.3 Program to convert a list of values into a set

```
s = set([1,2,'a','b',"def",4.56])
print(s)
```

OUTPUT

```
set(['a', 1, 2, 'b', 4.56, 'def'])
```

Programming Tip: If we add the same element multiple times in a set, they are removed because a set cannot have duplicate values.

The program given below demonstrates different ways of creating sets, that is, sets being created by using a list, tuples or string.

Example 6.4 Program to create a set

```
List1 = [1,2,3,4,5,6,5,4,3,2,1]
print(set(List1))  # list is converted into a set
Tup1 = ('a','b','c','d','b','e','a')
print(set(Tup1))# tuple is converted into a set
str = "abcdefabcdefg"
print(set(str))    # string is converted into a set
# forms a set of words
print(set("She sells sea shells on the sea shore".split()))
```

OUTPUT

```
set([1, 2, 3, 4, 5, 6])
set(['a', 'c', 'b', 'e', 'd'])
set(['a', 'c', 'b', 'e', 'd', 'g', 'f'])
set(['on', 'shells', 'shore', 'She', 'sea', 'sells', 'the'])
```

Like in case of mathematical sets, Python sets are also a powerful tool as they have the ability to calculate differences and intersections between other sets. Figures 6.4(a–d) given below demonstrate various set operations.

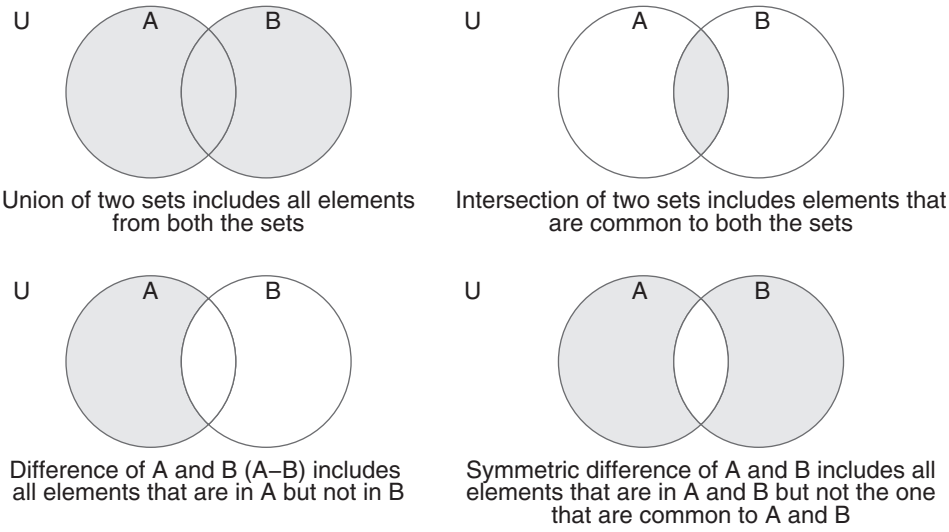

Figure 6.4 Set operations

Example 6.5 Program to find intersection, union, and symmetric difference between two sets

```
Coders = set(["Arnav","Goransh","Mani","Parul"])
Analysts = set(["Krish","Mehak","Shiv","Goransh","Mani"])
print("Coders : ", Coders)
print("Analysts : ", Analysts)
print("People working as Coders as well as Analysts :
", Coders.intersection(Analysts))
print("People working as Coders or Analysts :
", Coders.union(Analysts))
print("People working as Coders but not Analysts :
", Coders.difference(Analysts))
print("People working as Analysts but not Coders :
", Analysts.difference(Coders))
print("People working in only one of the groups :
", Coders.symmetric_difference(Analysts))
```

OUTPUT

```
Coders :  {'Arnav', 'Mani', 'Goransh', 'Parul'}
Analysts :  {'Shiv', 'Mani', 'Krish', 'Goransh', 'Mehak'}
People working as coders as well as Analysts :  {'Mani', 'Goransh'}
People working as Coders or Analysts :  {'Goransh', 'Parul', 'Krish', 'Mehak',
'Shiv', 'Arnav', 'Mani'}
People working as Coders but not Analysts : {'Arnav', 'Parul'}
People working as Analysts but not Coders : {'Shiv', 'Krish', 'Mehak'}
People working in only one of the groups : {'Krish', 'Mehak', 'Shiv', 'Arnav', 'Parul'}
```

Some other operations that can be performed on sets are discussed in Table 6.1. But before going through this table, just remember the following points.

Programming Tip: The `update()` method can take tuples, lists, strings, or other sets as its argument.

- Two sets are equal if and only if every element in each set is contained in the other.
- A set is less than another set if and only if the first set is a subset of the second set.
- A set is greater than another set if and only if the first set is a superset of the second set.

Table 6.1 Set Operations

Operation	Description	Code	Output

`s.update(t)`	Adds elements of set t in the set s provided that all duplicates are avoided	`s = set([1,2,3,4,5])`
		`t = set([6,7,8])`
		`s.update(t)`
		`print(s)`
			`(1,2,3,4,5,6,7,8)`

`s.add(x)`	Adds element x to the set s provided that all duplicates are avoided	`s = set([1,2,3,4,5])`
		`s.add(6)`
		`print(s)`
			`(1,2,3,4,5,6)`

`s.remove(x)`	Removes element x from set s. Returns KeyError if x is not present	`s = set([1,2,3,4,5])`
		`s.remove(3)`
		`print(s)`
			`(1,2,4,5)`

`s.discard(x)`	Same as `remove()` but does not give an error if x is not present in the set	`s = set([1,2,3,4,5])`
		`s.discard(3)`
		`print(s)`
			`(1,2,4,5)`

`s.pop()`	Removes and returns any arbitrary element from s. KeyError is raised if s is empty	`s = set([1,2,3,4,5])`
		`s.pop()`
		`print(s)`
			`(2,3,4,5)`

`s.clear()`	Removes all elements from the set	`s = set([1,2,3,4,5])`
		`s.clear()`
		`print(s)`
			`set()`

`len(s)`	Returns the length of set	`s = set([1,2,3,4,5])` `print(len(s))`	`5`
`x in s`	Returns True is x is present in set s and False otherwise	`s = set([1,2,3,4,5])` `print(3 in s)`	`True`
`x not in s`	Returns True is x is not present in set s and False otherwise	`s = set([1,2,3,4,5])` `print(6 not in s)`	`True`

`s.issubset(t)` or `s<=t`	Returns True if every element in set s is present in set t and False otherwise	`s = set([1,2,3,4,5])`
		`t = set([1,2,3,4,5,6,7,8,9,10])`
		`print(s<=t)`
			`True`

`s.issuperset(t)` or `s>=t`	Returns True if every element in t is present in set s and False otherwise	`s = set([1,2,3,4,5])`
		`t = set([1,2,3,4,5,6,7,8,9,10])`
		`print(s.issuperset(t))`
			`False`

Contd

Table 6.1 *Contd*

Operation	Description	Code	Output

`s.union(t)` or `s|t`	Returns a set s that has elements from both sets s and t	`s = set([1,2,3,4,5])`
		`t = set([1,2,3,4,5,6,7,8,9,10])`
		`print(s|t)`
			`(1,2,3,4,5,6,7,8,9,10)`

`s.intersection(t)` or `s&t`	Returns a new set that has elements which are common to both the sets s and t	`s = set([1,2,3,4,5])`
		`t = set([1,2,3,4,5,6,7,8,9,10])`
		`z = s&t`
		`print(z)`
			`(1,2,3,4,5)`

`s.intersection_update(t)`	Returns a set that has elements which are common to both the sets s and t	`s = set([1,2,10,12])`
		`t = set([1,2,3,4,5,6,7,8,9,10])`
		`s.intersection_update(t)`
		`print(s)`
			`(1,2,10)`

`s.difference(t)` or `s-t`	Returns a new set that has elements in set s but not in t	`s = set([1,2,10,12])`
		`t = set([1,2,3,4,5,6,7,8,9,10])`
		`z = s-t`
		`print(z)`
			`(12)`

`s.difference_update(t)`	Removes all elements of another set from this set	`s = set([1,2,10,12])`
		`t = set([1,2,3,4,5,6,7,8,9,10])`
		`s.difference_update(t)`
		`print(s)`
			`(12)`

`s.symmetric_difference(t)` or `s^t`	Returns a new set with elements either in s or in t but not both	`s = set([1,2,10,12])`
		`t = set([1,2,3,4,5,6,7,8,9,10])`
		`z = s^t`
		`print(z)`
			`(3,4,5,6,7,8,9,12)`

`s.copy()`	Returns a copy of set s	`s = set([1,2,10,12])`
		`t = set([1,2,3,4,5,6,7,8,9,10])`
		`print(s.copy())`
			`(1,2,12,10)`

`s.isisjoint(t)`	Returns True if two sets have a null intersection	`s = set([1,2, 3])`
		`t = set([4,5,6])`
		`print(s.isdisjoint(t))`
			`True`

`all(s)`	Returns True if all elements in the set are True and False otherwise	`s = set([0,1,2,3,4])` `print(all(s))`	`False`
`any(s)`	Returns True if any of the elements in the set is True. Returns False if the set is empty	`s = set([0,1,2,3,4])` `print(any(s))`	`True`

Contd

Table 6.1 *Contd*

Operation	Description	Code	Output

`enumerate(s)`	Returns an enumerate object which contains index as well as value of all the items of set as a pair	`s = set(['a','b','c','d'])`
		`for i in enumerate(s):`
		`  print(i,end= ' ')`
			`(0, 'a') (1, 'c') (2, 'b') (3, 'd')`

`max(s)`	Returns the maximum value in a set	`s = set([0,1,2,3,4,5])` `print(max(s))`	`5`
`min(s)`	Returns the minimum value in a set	`s = set([0,1,2,3,4,5])` `print(min(s))`	`0`
`sum(s)`	Returns the sum of elements in the set	`s = set([0,1,2,3,4,5])` `print(sum(s))`	`15`
`sorted(s)`	Return a new sorted list from elements in the set. It does not sorts the set as sets are immutable.	`s = set([5,4,3,2,1,0])` `print(sorted(s))`	`[0,1,2,3, 4,5]`

`s == t and s != t`	s == t returns True if the two set are equivalent and False otherwise. s!=t returns True if both sets are not equivalent and False otherwise	`s = set(['a','b','c'])`
		`t = set("abc")`
		`z = set(tuple('abc'))`
		`print(s == t)`
		`print(s!=z)`
			`True`
			`False`

Key points to remember

- A set cannot contain other mutable objects (like lists). Therefore, the following code will give an error when executed.

```
s = {10,20,[30,40]}
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 1, in <module>
    s = {10,20,[30,40]}
TypeError: unhashable type: 'list'
```

- To make an empty list you write, `List1 = [],` to make an empty tuple you write `Tup = (),` but to make an empty set you cannot write `s = {},` because Python will make this as a dictionary. Therefore, to create an empty set use the `set()` as shown below.

```
>>> s = set()
>>> print(s)
set()
print(type(s))
<class 'set'>
```

```
>>> t = {}
>>> print(type(t))
<class 'dict'>
```

- Since sets are unordered, indexing have no meaning.
- Set operations do not allow users to access or change an element using indexing or slicing. This is illustrated by code given below.

Example 6.6 Program to illustrate updating of a set

```
s = {1,2,3,4,5}
print(s[0])
```

OUTPUT

```
Traceback (most recent call last):
  File "C:\Python34\Try.py", line 2, in <module>
    print(s[0])
TypeError: 'set' object does not support indexing
```

Note A set can be created from a list but a set cannot contain a list.

- You can iterate through each item in a set using a `for` loop as shown in the code given below.

Example 6.7 Program to iterate through a set

```
s = set("Hello All, Good Morning")
for i in s:
    print(i,end = ' ')
```

OUTPUT

```
A   e d G i H M l o , g r n
```

- The `copy()` method makes a shallow copy of the set. This means that all the objects in the new set are references to the same objects as the original set.

Note To add a single element in the set use the `add()` method and to add multiple elements in the set, use the `update()` method.

Program 6.1 Write a program that generates a set of prime numbers and another set of odd numbers. Demonstrate the result of union, intersection, difference, and symmetric difference operations on these sets.

```
odds = set([x*2+1 for x in range(1,10)])
print(odds)
primes = set()
for i in range(2, 20):
    j = 2
    flag = 0
    while j<i/2:
        if i%j == 0:
```

```
            flag = 1
        j+=1
    if flag == 0:
        primes.add(i)
print(primes)
print("UNION : ",odds.union(primes))
print("INTERSECTION : ", odds.intersection(primes))
print("SYMMETRIC DIFFERENCE : ", odds.symmetric_difference(primes))
print("DIFFERENCE : ", odds.difference(primes))
```

OUTPUT

```
{3, 5, 7, 9, 11, 13, 15, 17, 19}
{2, 3, 4, 5, 7, 11, 13, 17, 19}
UNION :  {2, 3, 4, 5, 7, 9, 11, 13, 15, 17, 19}
INTERSECTION :  {3, 5, 7, 11, 13, 17, 19}
SYMMETRIC DIFFERENCE :  {2, 4, 9, 15}
DIFFERENCE :  {9, 15}
```

Program 6.2 Write a program that creates two sets. One of even numbers in range 1-10 and the other has all composite numbers in range 1–20. Demonstrate the use `all()`, `issuperset()`, `len()`, and `sum()` functions on the sets.

```
evens = set([x*2 for x in range(1,10)])
print("EVENS : ", evens)
composites = set()
for i in range(2, 20):
    j = 2
    flag = 0
    while j<=i/2:
        if i%j == 0:
           composites.add(i)
        j+=1
print("COMPOSITES : ", composites)
print("SUPERSET : ", evens.issuperset(composites))
print("ALL : ", all(evens))
print("LENGTH OF COMPOSITES SET : ", len(composites))
print("SUM OF ALL NUMBERS IN EVENS SET : ", sum(evens))
```

OUTPUT

```
EVENS :  {2, 4, 6, 8, 10, 12, 14, 16, 18}
COMPOSITES :  {4, 6, 8, 9, 10, 12, 14, 15, 16, 18}
SUPERSET :  False
ALL :  True
LENGTH OF COMPOSITES SET :  10
SUM OF ALL NUMBERS IN EVENS SET :  90
```

Program 6.3 Write a program that creates two sets—squares and cubes in range 1–10. Demonstrate the use of `update()`, `pop()`, `remove()`, `add()` and `clear()` functions.

```
squares = set([x**2 for x in range(1,10)])
cubes = set([x**3 for x in range(1,10)])
print("SQUARES : ", squares)
print("CUBES : ", cubes)
squares.update(cubes)
print("UPDATE : ", squares)
squares.add(11*11)
squares.add(11*11*11)
print("ADD : ", squares)
print("POP : ", squares.pop())
squares.remove(1331)
print("REMOVE : ", squares)
squares.clear()
print("CLEAR : ", squares)
```

OUTPUT

```
SQUARES :  {64, 1, 4, 36, 9, 16, 49, 81, 25}
CUBES :  {64, 1, 512, 8, 343, 216, 729, 27, 125}
UPDATE :  {64, 1, 512, 4, 36, 8, 9, 16, 49, 81, 729, 343, 216, 25, 27, 125}
ADD :  {64, 1, 512, 121, 4, 36, 8, 9, 16, 49, 81, 1331, 729, 343, 216, 25, 27, 125}
POP :  64
REMOVE :  {1, 512, 121, 4, 36, 8, 9, 16, 49, 81, 729, 343, 216, 25, 27, 125}
CLEAR :  set()
```

Program 6.4 Write a program that has a list of countries. Create a set of the countries and print the names of the countries in sorted order.

```
countries = ['India', 'Russia', 'China', 'Brazil', 'England']
C_set = sorted(set(countries))
print(C_set)
```

OUTPUT

```
['Brazil', 'China', 'England', 'India', 'Russia']
```