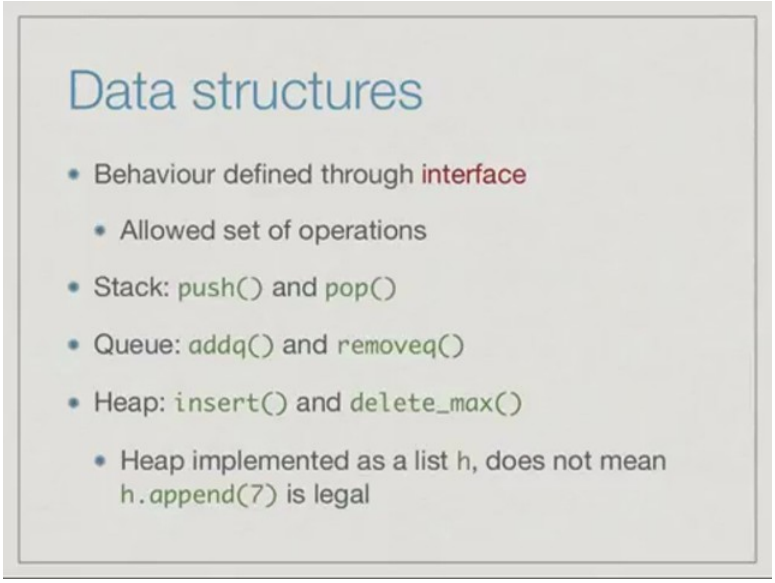
Programming Data Structures and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 07
Lecture - 01
Abstract Datatypes, Classes and Objects

(Refer Slide Time: 00:02)



We have seen how to implement data structures such as, stacks, queues and heaps using the built in list type of Python. It turns out that one can go beyond the built in types in python and create our own data types. So, we will look at this in more detail in this weeks' lectures.

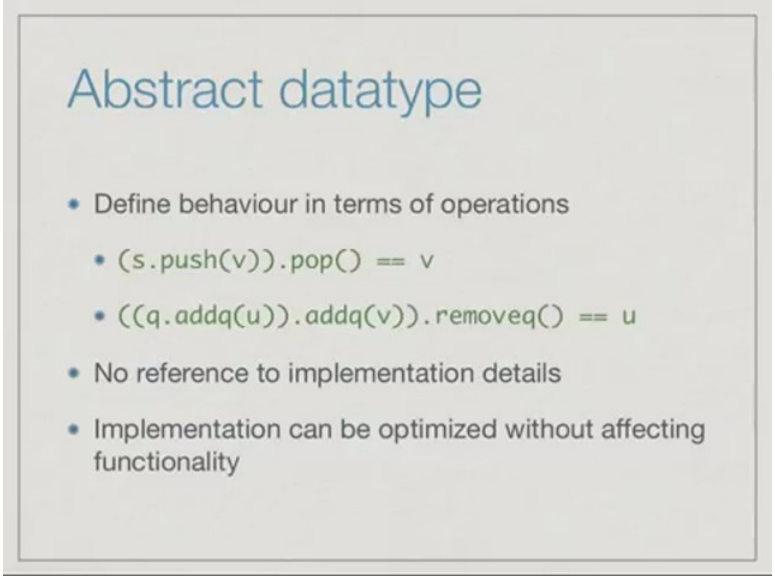
Let us revisit what we lean by a data structure. A data structure is basically an organization of information whose behavior is defined through an interface. So an interface is nothing but the allowed set of operations, for instances for a stack the allowed set of operations are push and pop. And of course, we can also query whether a stack is empty or not.

Likewise, for a queue the only way we can modify a queue is to add something to the tail of the queue using the function add q and remove the element at the head of the queue

using the function remove q. And for a max heap for instance, we have the functions insert to add an element and delete max which removes the largest element from the heap.

Now, just because we implement a heap as a list it does not mean that the functions that are defined for lists are actually legal for the heap. So if we have a heap h, which is implemented as a python list though the list will allow an append function. The append function on it is own does not insert a value and maintain the heap property. So, in general the call such as h dot append 7 would not be legal.

(Refer Slide Time: 01:35)



So, we want to define new abstract data types in terms of the operations allowed. We do not want to look at the implementation and ask whether it is a list or not, because we do not want the implementation to determine what is allowed, we only want the actual operations that we define as the abstract interface to be permitted.

For instance if we have a stack s and we push value v then the property of a stack guarantees that if we immediately apply pop the value we get back is our last value push and therefore we should get back v. In other words, if we execute this sequence we first to s dot push and then we do a pop then the value that we pushed must be the value that

we get back.

This is a way of abstractly defining the property of a stack and how push and pop interact without actually telling us anything about how the internal values are represented. In the same way if we have an empty queue and we add to it two elements u and v and then we remove the head of the queue, then we expect that we started with an empty queue and then we put in from this end u and then we put in a v, then the element that comes out should be the first element namely u. In other words assuming that this is empty then if we add u and add v and then remove the head we should get back first element that we put in namely u.

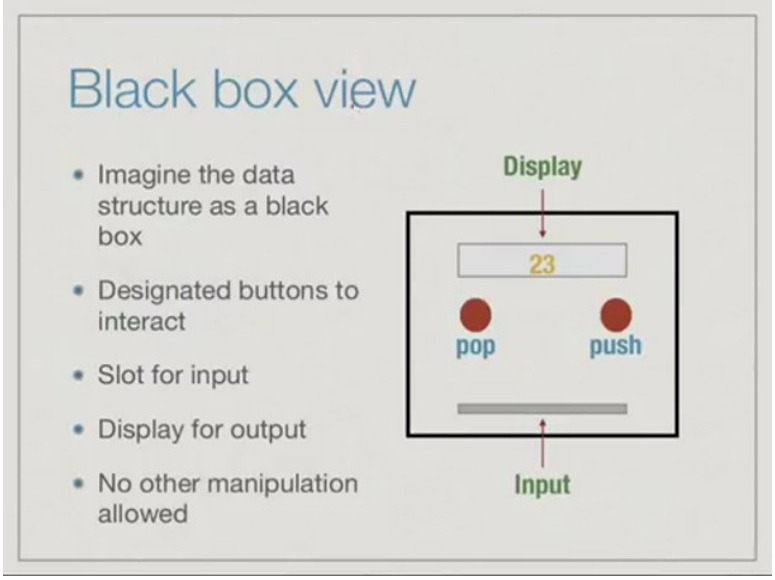
The important thing is that we would like to define the behavior of a data type without reference to the implementation. Now this can be very tedious because you have to make sure that it capture all the properties between functions, but this can be done and this is technically how an abstract data type is defined. Now large purposes we will normally define it more informally and we will make reference to the implementation, but we definitely do not want the implementation to determine how these functions work.

In other words, we should be able to change one implementation to another one such that the functions behave the same way and the outside user has no idea which implementation is used. Now this is often the case when we need to optimize implementation, we might come up with an inefficient implementation and then optimize it. For instance we saw that for a priority queue we could actually implement it as a sorted list and then we could implement insert as an insert operation in a sorted list which you take order n time, but delete max would just remove the head of the list.

This is not optimal because over a sequence of n inserts and deletes this takes time order n square. So if we replace the internal implementation from a sorted list to a heap we get better behavior, but in terms of the actual values that the user sees as a sequence of inserts and delete max the user does not see any difference between the sorted list implementation and the heap implementation. Perhaps, there is a perception that the one is faster than the other, but the actual correctness of the implementation should not be affected by how you choose to represent the data. So, this is the essence of defining an

Abstract datatype.

(Refer Slide Time: 04:40)



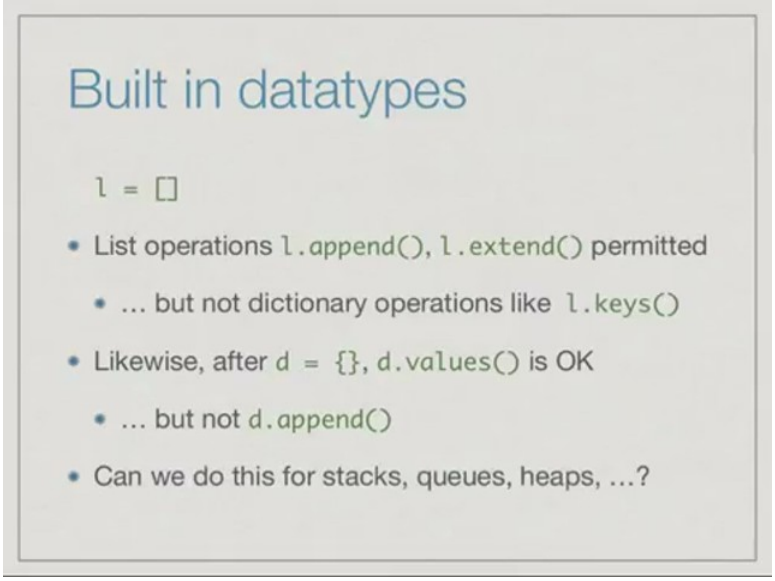
So, good way to think of an abstract datatype is as a black box which allows limited interaction. Imagine something like an ATM machine. So, we have the data structure as a black box and we have certain buttons which are the public interface, these are the functions that we are allowed to use. In this picture imagine this is a stack and the buttons were allowed to push are pop and push let they are allowed to remove the top elements from the stack; they are allowed to put an element into the stack.

Now this requires us to also add and view things from the stack, so we also have a slot for input which is shown as a kind of a thing at the bottom here we have the slot for input. And we have the way to receive information about the state of the stack. So we can imagine that we have some kind of a display.

This is typically how we would like to think of a data structure, we do not want to know what is inside the black box we just want to specify that if we do a sequence of button pushes and we start supplying input through the input box what do we expect to see if the display. Other than this, no other manipulation should be allowed. We are not allowed to exploit what is inside the box in order to quickly get access say to the middle of a stack

or the middle of a queue. So we do not want such operations, we only want those operations which the externally visible interface or the buttons in this case of the black box picture allow us to use.

(Refer Slide Time: 06:21)



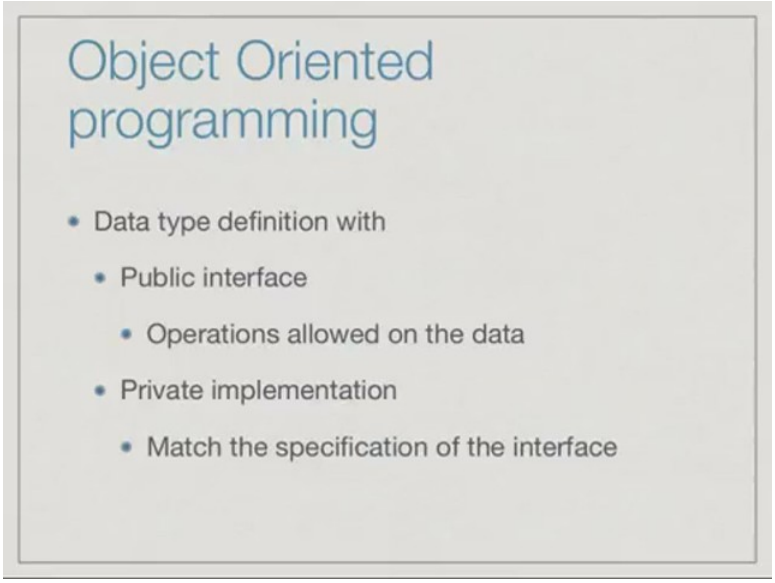
In a sense this is already implemented when we use the built in data types of python, if we announce that the name l is of type list by setting l to the empty list then immediately python will allow us to use operations like append and extend on this list, but because it is of list type and not dictionary type we would not be able to execute an operations such as keys which is defined for dictionaries are not list.

Likewise if we define d to be an empty dictionary then we can use a function such as d dot values to get the list of values currently stored, but we cannot manipulate d as a list. So, we cannot say d dot append it will give us an error. Python uses the type information that it has about the value give assign to a name to determine what functions are legal which is exactly what we are trying to do with these abstract data types. We are trying to say that the data type on it is own should allow only certain limited types of access whose behavior is specified without telling us anything about the internal implementation.

Remember for instance we saw that in a dictionary even if we add a sequence or values in the particular order we ask for the values after sometime they may not written in the same order, because internally there is some optimization in order to make it fast to look up a value for it. We have no idea actually how dictionaries implemented inside, but what we do know is that if we provide a key and that key is a valid key we will get the associated with that key, we do not ask how this is done and we do not know whether from one version of python to the next the way in which this is implemented changes.

Our question is, that instead of using the built in list for stacks, queues and heaps and other data structures can we also defined a data type in which certain operations are permitted according to the type that we start with.

(Refer Slide Time: 08:17)

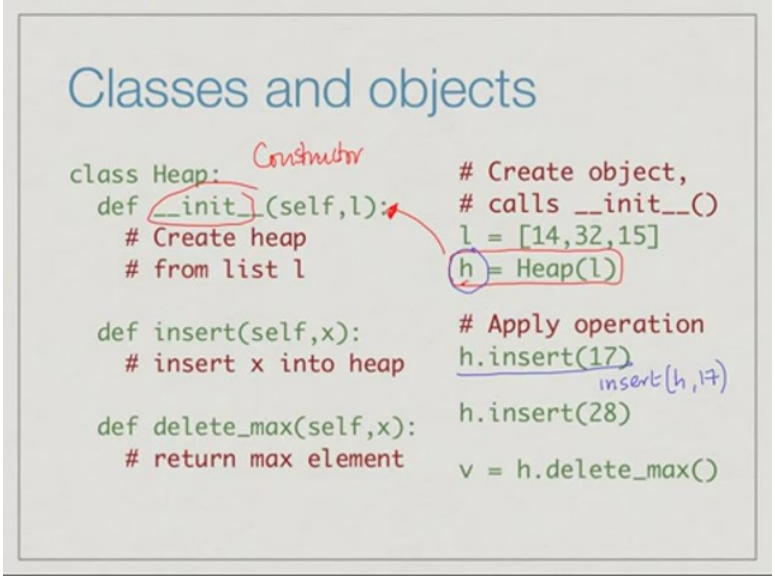


This is one of the main things which are associated with a style of programming called Object Oriented program. In object oriented program, we can provide data type definitions for new data types in which we do precisely what we have been talking about we describe the public interface, that is the operations that are allowed on the data and separately we provide an implementation which should be private, we will discuss later that in python we do not actually have a full notion of privacy because of the nature of the language.

But ideally the implementation should not be visible outside only the interface should allow the user to interact with the implemented data. Of course, the implementation must be such that the functions that are visible to the user behave correctly.

So here for instance if we had a heap the public interface would say insert and delete max, the private implementation may be a sorted list or it may be a heap and then we would then have to ensure that if we are using a sorted list we implement delete max and insert in the correct way and if we switch from that to a heap the priority queue operations remain the same.

(Refer Slide Time: 09:33)



In the terminology of object oriented programming there are two important concepts; Classes and Objects. A class is a template very much like a function definition is a template, when we say def and define a function the function does not execute it just gives us a blue print saying that this is what would happen if this function were called with a particular argument and that argument to be substituted for the formal parameter in the function and the code in the function will be execute to the corresponding value.

In the same way a class sets up a blue print or a template for a data type. It tells us two things it tells us; what is the internal implementation? How is data stored? And it gives

us the functions that are used to implement the actual public interface. So, how you manipulate the internal data in order to effect the operations that the public interface allows. Now once we have this template we can construct many instances of it. So, you have the blue print for a stack you can construct many independent stacks, each independent stack has it is own data that stacks do not interfere with each other.

Each of them has a copy of the function that we have defined associated with it. Rather than the kind of the main difference from classical programming is, in classical programming you would have for instance a function like say push define and it will have two parameters typically a stack and a value. So, you have one function and then you provide it the stack that you want to manipulate.

On the other hand, now we have several stacks s1, s2, s3, etcetera which are created as instance as class, and logically each of them has it is own push function. So there is a push associated with s1, that the push associated is s2, the push associated with s3 and so on. Each of them is a copy of the same function derived from this template, but this implicitly attach to the single object. So, this is just a slight difference in perspective instead of having a function to which you pass the object that you want to manipulate you have the object and you tell it what function to apply to itself.

So, let us look at a kind of example this would not be a detailed example it will just give you a flavor of what we are talking about. Here is a skeleton of a definition of a class heap. So now, we instead of using the built in list we want to define our own data type heap. So there are some function definitions. These def statements and these correspond to definition in the functions and what we will see is that inside these definitions we will have values which are stored in each copy of the heap. So, just to get a little bit of an idea about how this is would work.

When we create on object of type heap we call it like a function. So, we say h is equal to heap l, so this implicitly says give me an instance of the class heap with the initially value l passed to it now this calls this function init which is why it is called init. So, init is what is called a constructor. A constructor is a function that is called when the object is created and sets it up initially in this particular case our constructor is presumed to take

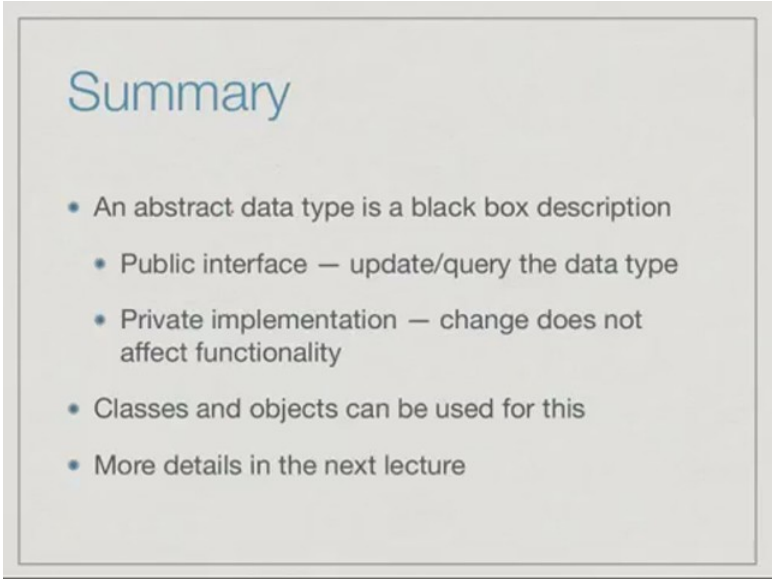
an arbitrary list of values say 14, 32, 15 and heapify it. So, somewhere inside the code of init there will be a heapification operation which we are not actually shown in this slide.

This is how you create objects. You first define a class we will look at a complete example soon, we define a class and then you call the class sort of like a function and the name that is attach to this function call or this class were becomes a new object, As we said we have functions like insert and delete max define for heaps, but it is like we have the separate copy of this function for each individual heaps.

In this case we are created a heap h, so we want to tell h insert in yourself the value 70. So, we write that as insert with respect to h. So, h dot insert 17, as suppose to insert h 17 which would be the normal functional style of writings. We would normally pass it the heap and the value, here instead we say given the heap h apply to the heap h the function insert with the argument 70.

The next line says apply to the heap h in function insert to the value 28 and then for instance we can now ask h to return the maximum value by it is an h dot delete max and store the return value in the main v.

(Refer Slide Time: 14:28)



So, what we would like to emphasize is that an abstract data type should be seen as a black box. Like a black box has a public interface the buttons that you can push to update and query the data type to add things, delete things, and find out what whether the data type is empty and so on. Inside we have a private implementation. This actually stores data in some particular way to make the public functions work correctly. But the important thing is, changing the private implementation should not affect how the functions behave in terms of input and output.

They may behave differently in terms of efficiency, you might see that one version is faster than another or one version slower than another, but this is not the same as saying that the functions change. So, we do not want the values to change, if we have a priority queue and we insert a set of values and then delete max no matter how the priority queue is actually implemented internally the delete max should give us the same value at the end.

So, we saw that python supports object oriented programming, we shall look at it in more detail in the next couple of lectures in these weeks course, but the main concept associated with this objected oriented programming are classes and objects. Classes are templates for data types and objects are instances of these classes they have a concrete data types which we use in our program.