Python list[] and the ArrayList Data Structure

CSC 231 – Introduction to Data Structures

Dr. Lucas Layman

UNCW

Data types vs. data structures

Data type: data and operations you can perform on that data, e.g., the Python `list` class

Data structure: an implementation of how the data items are stored "under the hood", i.e., how the Python `list` class organizes its data internally.

The ArrayList data structure

Python's `list` class organizes its data in what we call an ArrayList structure.

An *array* is a contiguous block of memory.

The main benefit of an array-based list is that we can *access* any element of the list in O(1) time.

Address	Value
0x0000000	
0x0000008	
0x0000010	
0x0000018	
0x0000020	
0x0000028	
0x0000030	

What happens when a `list` is instantiated in memory

```
names = []
```

`HEADER` is 24 bytes

`length` is 8 bytes

- the exact number of bytes will depend on your computer, but it will be *constant* for your computer and that is the important thing.

the `length` is returned when you call len(names).

Python *pre-allocates* a larger block of memory for the list in anticipation you will add data to it

Address	Value
0x0000000	<OBJECT HEADER>
0x0000008	
0x0000010	
0x0000018	length: int
0x0000020	
0x0000028	
0x0000030	
0x0000038	
0x0000040	

`names` object

free memory

list.append()

```
names.append('Alice')
```

Appending an item to the list puts a *reference* to the memory address of the item's value at the end of the list's memory space

Each object reference is 8 bytes

- the exact number of bytes will depend on your computer, but it will be *constant*

Address	Value
0x0000000	<OBJECT HEADER>
0x0000008	
0x0000010	
0x0000018	length: int
0x0000020	0x2dbf9a0
0x0000028	
0x0000030	
0x0000038	
0x0000040	

names object

'Alice'

free memory

This is the *data structure* of a Python list

```
names.append('Bob')
```

A data structure is how a data type organizes its data internally.

The Python `list`'s data structure is to use a contiguous array of memory to store references to items

Address	Value
0x0000000	<OBJECT HEADER>
0x0000008	
0x0000010	
0x0000018	length: int
0x0000020	0x2dbf9a0
0x0000028	0x42e7599
0x0000030	
0x0000038	
0x0000040	

names object

'Alice'

'Bob'

A simplified model of the `list` data structure

```
names = []
names.append('Alice')
names.append('Bob')
names.append('Fran')
names.append('John')
names.append(12345)
names.append(10.0)
```

names

len	6
0	'Alice'
1	'Bob'
2	'Fran'
3	'John'
4	12345
5	10.0

REMEMBER: Lists store object references, not the data value!

Python `list`'s data structure dictates the $O(f(n))$ of its operations

Big-O	Code	Operation
O(1)	list[index]	get item by index
O(1)	list[index] = x	set object at index to be x
O(1)	list.append(x)	append x to the end of list
O(1)	list.pop()	remove from end of list and return it
O(n)	list.pop(i)	remove the item at index i and return it
O(n)	list.remove(x)	remove the first item whose value is x
O(n)	list.insert(i, x)	insert x at index i
O(n)	iteration (for x in list)	iterate over each item in list

You need to know these.

A closer look at Python list[] operations

Refer to your worksheet

You are responsible for knowing *how* the operations from the previous slide work on the ArrayList data structure underlying Python's list[] data type

You are responsible for knowing the Big-O of each operation

Demonstrating the implications of Big-O in code

We will test the runtimes of list[] operations to see firsthand O(1) vs. O(n) time complexity

Abstract Data Types (ADTs), Data Types, and Data Structures (oh my…)

CSC 231 – Introduction to Data Structures

Dr. Lucas Layman

UNCW

Python concept: `return None`

```
def get_index(item, a_list):
    index = 0
    for x in a_list:
        if item == x:
            return index
        index += 1

nums = [1, 981273, 132, 120984, 6, 5, 7456, 345, 23, 4]
print(get_index(120984, nums))
print(get_index(99999, nums))
```

Python concept: `return None` (2)

What is None?

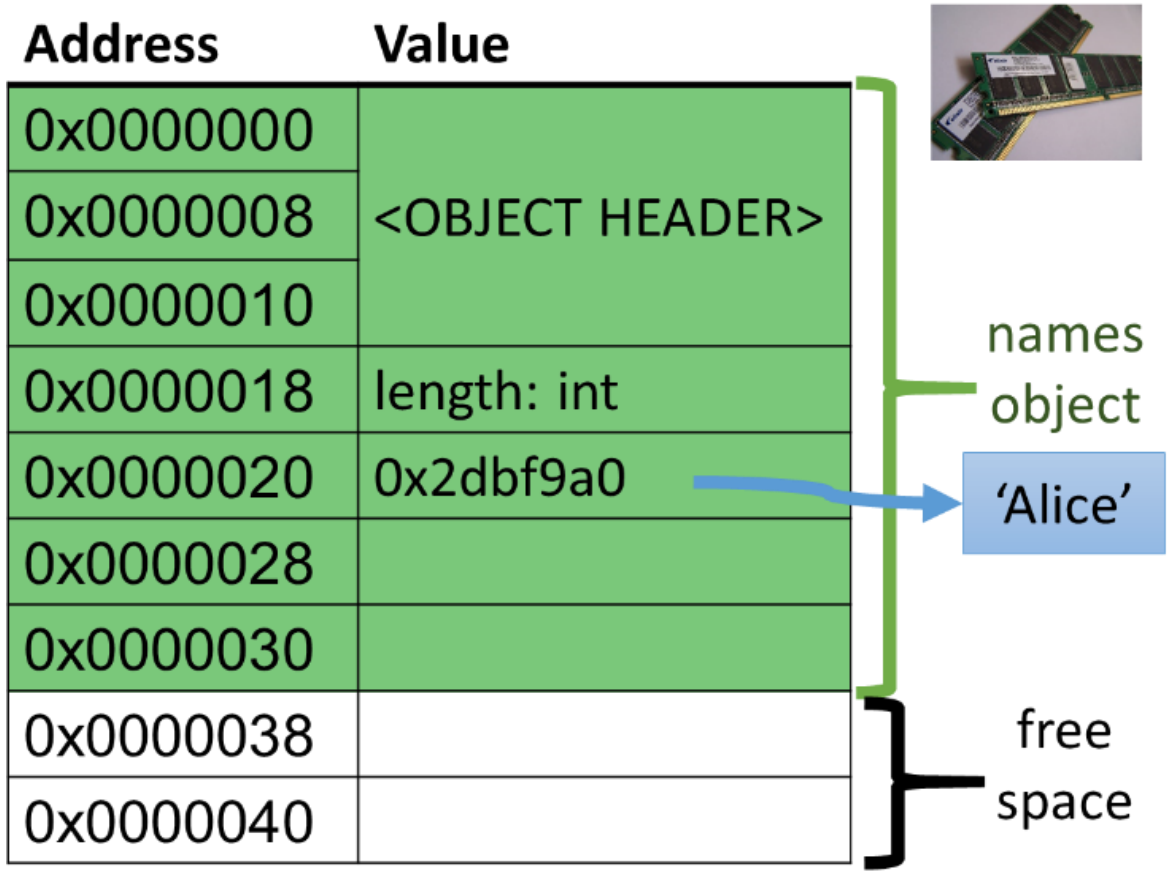
- A built-in constant, like `True` or `False`
- None is used to represent the *absence of a value*

Python functions and methods return None *by default* if you do not specify a return value

Python `list` is implemented with an ArrayList data structure

Big-O	Code
O(1)	list[index]
O(1)	list[index] = x
O(1)	list.append(x)
O(1)	list.pop()
O(n)	list.pop(i)
O(n)	list.remove(x)
O(n)	list.insert(i, x)
O(n)	iteration (for x in list)

You need to know these.



ADTs, data types and data structures

Abstract Data Type (ADT): a conceptual thing across computing. Usually oriented around solving a problem.

- A collection of data items (its state), and
- The basic relationships among them an operations that must be performed on them (its behavior)

Data type or *type* or *class*: The implementation of an ADT using a language

Data structures: How the data type organizes its data items

Your first Abstract Data Type

The *UnorderedList* ADT: a collection of items where each item holds a relative position with respect to the others

It is *unordered* because the position of an item is *not* determined by its value

UnorderedList, list[], and ArrayList

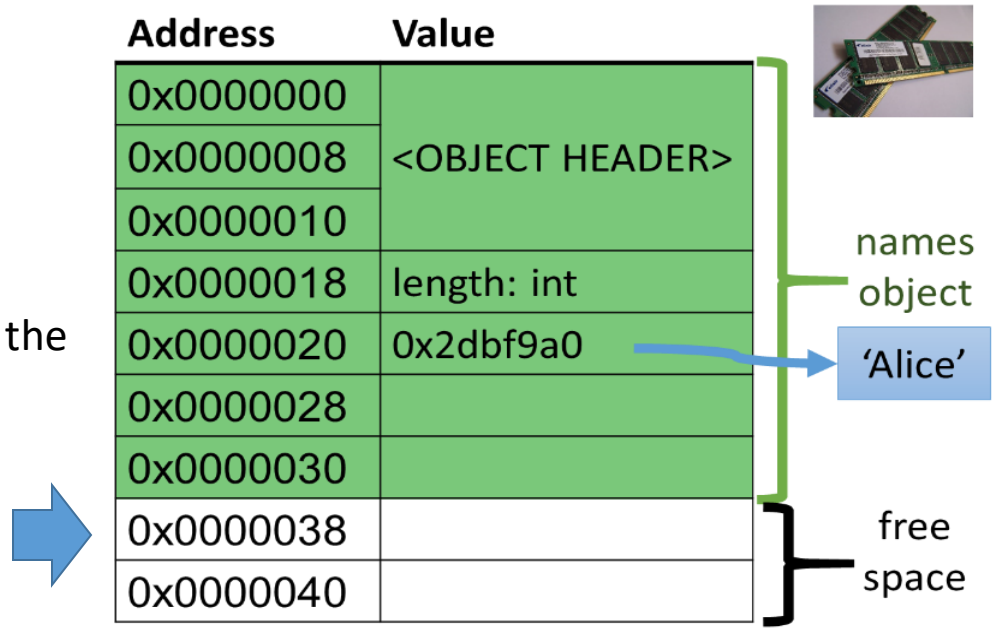
UnorderedList *ADT* operations:

- adding and remove items by value
- appending items to the end
- inserting and removing an item at a specific location (index)
- determining the size of the list, or if it is empty
- searching for an item in the list

The Python `list[]` *data type* is a concrete implementation of the UnorderedList ADT.

- We will create another data type next week that also implements the UnorderedList ADT

The Python list[] data type uses an ArrayList *data structure* to organize the elements



More ADTS – the Stack, the Queue, and the Deque

What distinguishes one ADT from another is the *location* in which these additions and removes occur

A *stack* adds and removes from the top only

A *queue* adds to the rear and removes from the front

A *deque* adds and removes from either end

Stacks

CSC 231 – Introduction to Data Structures

Dr. Lucas Layman

UNCW

Stacks

A *stack* or "push-down stack" is an ADT where:

Addition and removal of items always happens at the *same end*, which is called the *top* of the stack

A stack has *last-in first-out (LIFO)* behavior – the most recently added item will be removed first.

Stack properties and applications

The LIFO nature of stacks means that items are removed in the reverse order in which they were added

Some applications of stacks:

- The 'Back' button in your browser
- Undo
- Parsing programming language syntax

Stack data items and behaviors

Stacks have the following methods:

- `Stack()` – class constructor
- `push(item)` – adds item to the top of the stack.
- `pop()` – removes and returns the top item on the stack
- `peek()` – returns the top item on the stack, but does not remove it.
- `is_empty()` – returns True if the stack is empty, False otherwise
- `size()`, a.k.a., `__len__()` – returns the numbers of item on the stack

Stack data is typically stored in some sort of list

Visualizing a stack

Refer to your worksheet

Coding a Stack

Python concept: modules

.py scripts are technically called Python *modules*.

Importing a module allows you to use its code (main statements, functions, classes) in another module

- e.g., you may have seen `import sys` or `import random` at the top of files.

Python concept: modules (2)

Two approaches to using module contents. Either is fine:

```
import stack                    # there is a file named stack.py
x = stack.Stack()               # the Stack class in is stack.py
x.push("Alice")

from stack import Stack
x = Stack()
x.push("Alice")
```

Python concept: modules (3)

When you import a module, ALL of the code in the module runs.

You may wish to specify that some parts of your module only run when that module is the "main" program.

The code in `if __name__ == "__main__":` will run *only* when the module is run as the "main" program but will *not* run when imported.

Big-O for stack operations

	"Top" of the stack is the ***end*** of the list	"Top" of the stack is ***index 0*** of the list
Stack()	O(1)	O(1)
push(item)	O(1)	O(n)
pop()	O(1)	O(n)
peek()	O(1)	O(1)
is_empty()	O(1)	O(1)
size()	O(1)	O(1)

Queues

CSC 231 – Introduction to Data Structures

Dr. Lucas Layman

UNCW

Queues

A *queue* is an ADT where:

Addition happens at one end: the *rear*

Removal happens at the other end: the *front*

A queue has *first-in first-out (FIFO)* behavior – the first item added will be removed first.

Queue properties and applications

The FIFO nature of queues means that the last item added must wait for all the others to be removed

Some applications of queues:

- A printer queue
- Processing keystrokes

Queue data items and behaviors

Queue data items are typically stored in a *list*

Queues have the following methods:

- `Queue()` – class constructor
- `enqueue(item)` – adds item to the rear of the queue
- `dequeue()` – removes and returns the item at the front of the queue
- `is_empty()` – returns True if the queue is empty, False otherwise
- `size()`, a.k.a., `__len__()` – returns the numbers of items in the queue

Visualizing a queue

Refer to your worksheet

Big-O for queue operations

	"Front" of the queue is the ***end*** of the list	"Front" of the queue is ***index 0*** of the list
Queue()	O(1)	O(1)
enqueue(item)	O(n)	O(1)
dequeue()	O(1)	O(n)
is_empty()	O(1)	O(1)
size()	O(1)	O(1)

We ***cannot*** get all of these to be O(1) if we use a single Python list!

Deque – double-ended queue

CSC 231 – Introduction to Data Structures

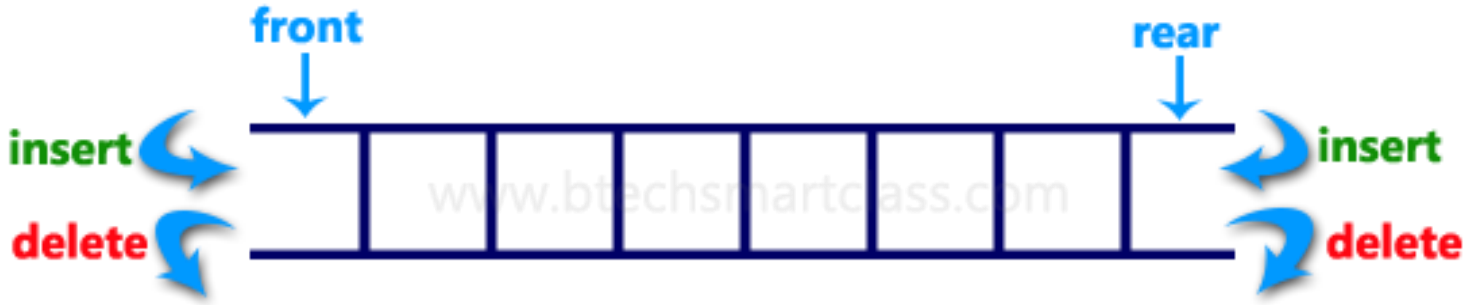
Dr. Lucas Layman

UNCW

Deque ("deck") – double-ended queue

A de*que* is an ADT where addition and removal happen at both ends.

A deque essentially combines the behaviors of a stack and a queue



Deque properties and applications

The deque provides maximum flexibility in where items are added to the list, but is also least efficient

Some applications of deques:

- CPU process scheduling
- Browser history or Undo lists that are limited in size, i.e., your browser can remember only the last 20 sites you visited

Deque data items and behaviors

Deque data items are typically stored in a *list*

Deques have the following methods:

- `Deque()` – class constructor
- `add_front(item)` – adds item to the front of the deque
- `add_rear(item)` – adds item to the rear of the deque
- `remove_front()` – removes and returns the item at the from of the front of the deque
- `remove_rear()` – removes and returns the item at the from of the rear of the deque
- `is_empty()` – returns True if the queue is empty, False otherwise
- `size()`, a.k.a., `__len__()` – returns the numbers of items in the deque

Visualizing a deque

Refer to your worksheet

Let's talk Big-O for deques

	"Front" of the deque is the ***end*** of the list	"Front" of the deque is ***index 0*** of the list
Deque()	O(1)	O(1)
add_front(item)	O(1)	O(n)
add_rear()	O(n)	O(1)
remove_front()	O(1)	O(n)
remove_rear()	O(n)	O(1)
is_empty()	O(1)	O(1)
size()	O(1)	O(1)

We ***cannot*** get all of these to be O(1) if we use a single Python list!

Improving Queue and Deque operations

The primary behaviors of queues and deques that we care about are adding and removing items…

But in both of these ADTs, at least one of these operations (adding or removing) is O(n) because we use a Python list internally

We can make the adding and removing operations O(1), but we need to create our own notion of a list… next time