Representing Sequences with Lists

Everything in Python is an Object

```
x=3
y=3
z=y
y=4
```

The above code is implemented as follows in Python:

- 3 is an integer object, stored in memory
- x is a variable which references that object
- y is a variable which references the same object – only 1 copy of a basic type
- z is a variable which references the same object
- y is then changed to point to a different object

This is the same for bools, floats, etc.

User-defined Classes and Objects

Every time we create an object from a class, Python creates a new object in memory:

```
c1 = Card(3, 3)
c2 = Card(3, 3)
c3 = c2
c2 = Card(6, 1)
```

Here however, c1 and c2 are pointing to separate objects in memory, not both pointing to the same object.

Python Lists Are Array-Based Lists

A list in Python is a sequence of references to objects. The references in the list are maintained in a contiguous physical block in memory, packed one after another in sequence – this is called an array-based list.

A list of ints is a sequence of references to int objects.

Tuples

A tuple is a sequence of references to objects. The length and content of a tuple cannot change.

The implementation is similar to lists, but it can take advantage of the fact that they can't change.

Space Required for a List

Since a list is a sequence of references, the amount of space required for a list depends on the number of elements and not the type of the elements.

Sequence Operations

Accessing Elements by Index

Since Python knows how much space each entry in a list takes regardless of what's in the list and knows the location in memory of the start of the list, accessing an element by index only takes this computation:

- start address + size * index

So accessing elements by index takes constant time. It is O(1).

Searching for an Element

```
for x in mylist:
    if x == 4:
        return True
return
```

The condition `if 4 in mylist` in Python runs an algorithm equivalent to the one above (though written in another language). For objects it's the same:

```
c1 = Card(3, 3)
for card in mylist:
    if card.is_equals(c1):
        return True
return False
```

This operation is O(n).

Appending to a List

The list has space for each reference set aside in memory. What happens when we increase the size of the list?

In the best case, Python will create the space at the end and put the reference there.

If that space is currently occupied by other data, Python will find a new space in memory and put the whole list there. Python needs to keep all the data together in order to find elements by index as we saw.

So appending items to a list may require reservation of a new area in memory for the entire list, and time to copy the list elements across.

Naive Append

```
mylist = []
for i in range(1000):
    mylist.append(i)
```

In the worst case (the whole list has to be moved each time a new item is appended), this will take `0.5 * 999 * 1000` copy operations, which is 499500 copy operations.

This is O(n squared), where n is the size the list may grow to.

This is not what Python does. A list of length 0 takes up 36 bytes, length 1–4 takes 52 bytes, length 5–8 takes 68 bytes, and so on.

When asked to append, if the list is full, Python creates a bigger list with size based on the current list.

Complexity of List Doubling

Policy: each time we must grow the list, double its size. Suppose we are unlikely, and must copy the list each time we grow it.

To create a list that reaches size n, this will take:

- 1 assignment +
- 1 copy operation + 1 assignment +
- 2 copy operations + 2 assignments +
- 4 copy operations + 4 assignments +
- …
- n/2 copy operations + n/2 assignments

This is n copies and n assignments. Making a list of size 1000 takes 2000 assignments in the worst case, while the naive method took 500000 assignments.

Doubling the size each time space is needed takes O(n) to build a list of size n.

Average cost of a single append is O(1).

Popping from a List

Popping is taking an element out of a list and using it. Python's lists have no spaces in memory, so the list must close up. Every item after the popped item must be moved forward.

For a list of length n, pop(i) takes (n - 1 - i) copies. Pop is O(n).

Memory Space Management

Pop and other operations all create empty cells at the end of the list. Python will manage the space, and when the proportion of empty space exceeds a limit, Python will release the space for other uses.