# Array-based Lists

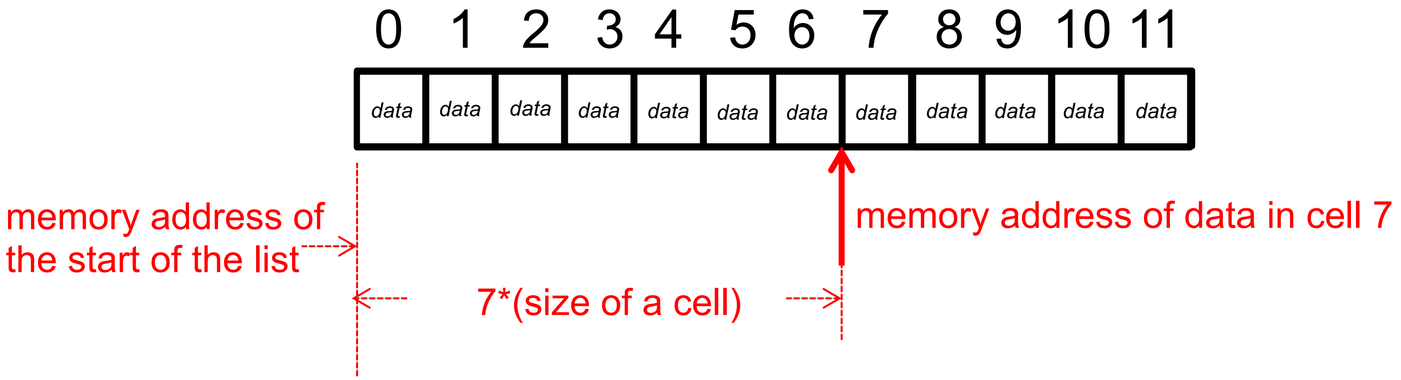
Storing data in memory
Storing data in lists (in memory)
Python's sequence data types

Complexity of standard list operations in Python

# Arrays

An *array* is a single block of memory of fixed size (determined when it is created), conceptually divided into cells packed one after another with no gaps, where we store data in each cell. Standard in many languages.

Each cell is exactly the same size. In most languages, the data stored in the cells must all be of the same data type.


0 1 2 3 4 5 6 7 8 9 10 11
data data data data data data data data data data data data
memory address of the start of the list
memory address of data in cell 7
7*(size of a cell)

If we know the address of the start of the array, and we know the size of each cell, we can use simple arithmetic to jump to any numbered cell.

Address of cell *i* = address of cell *0* + (*i* **size)*

O(1) time to access a cell by index

# Everything in Python is an object

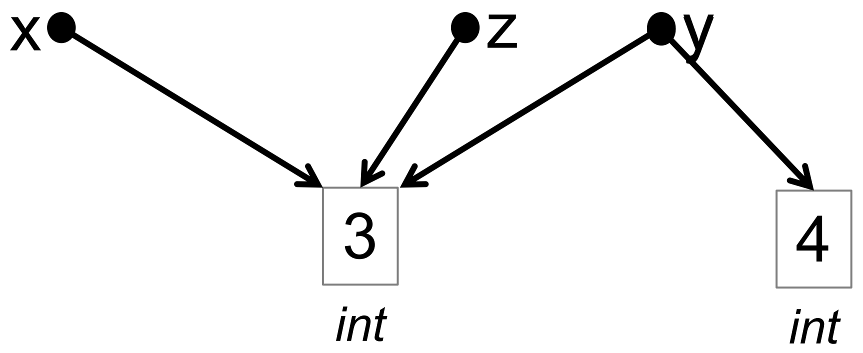
```
x=3
y=3
z=y
y=4
```

The reference maintains the memory address of the object.

is implemented as follows:

- 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

same for
bool
float



# Playing cards

A standard deck of cards has 4 suits, and 13 cards per suit,
2, 3, ..., 9, 10, Jack, Queen, King, Ace.

Define a class to represent a card.
Each object (member of class) will have a specific suit, and a specific rank.

```
Class: Card
Fields: rank: {2,3,...,14}
        suit: {1,2,3,4}
Methods: ...
```

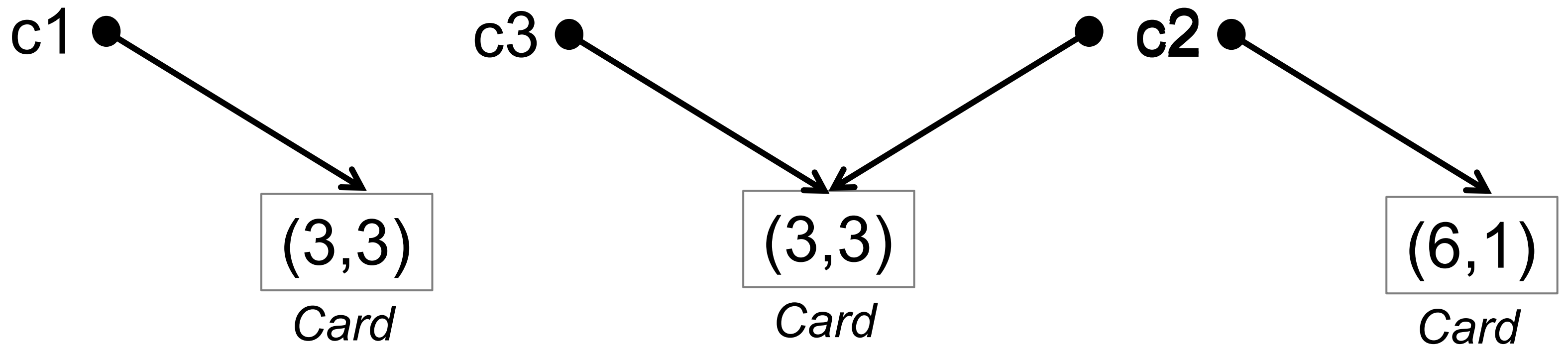
```
Class: Hand
Fields: cardlist:
            <list of Card objects>
Methods: ...
```

# User-defined classes and objects

Each time we create an object from a class, Python creates a new object in memory (Python decides where it is stored):

```
c1 = Card(3,3)
c2 = Card(3,3)  #creates a different object, same values
c3 = c2
c2 = Card(6,1)
```

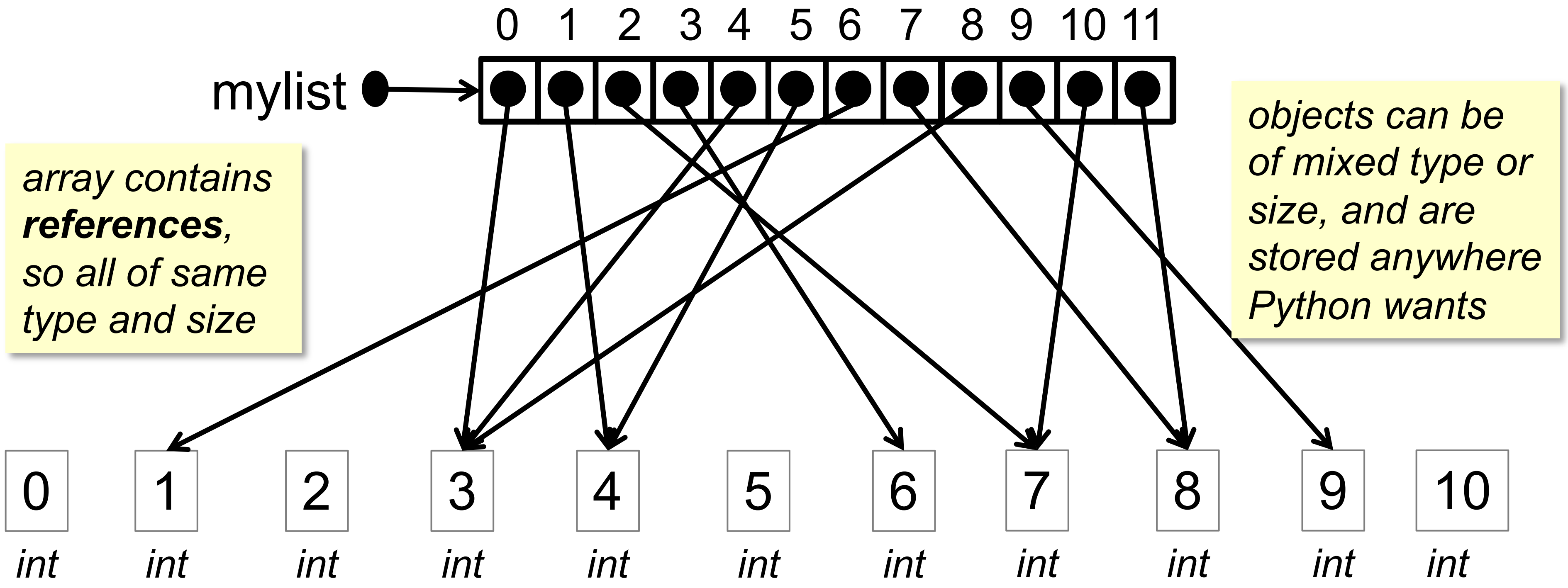
Note the difference to the previous slide: Card(3,3) is a call to create a new object

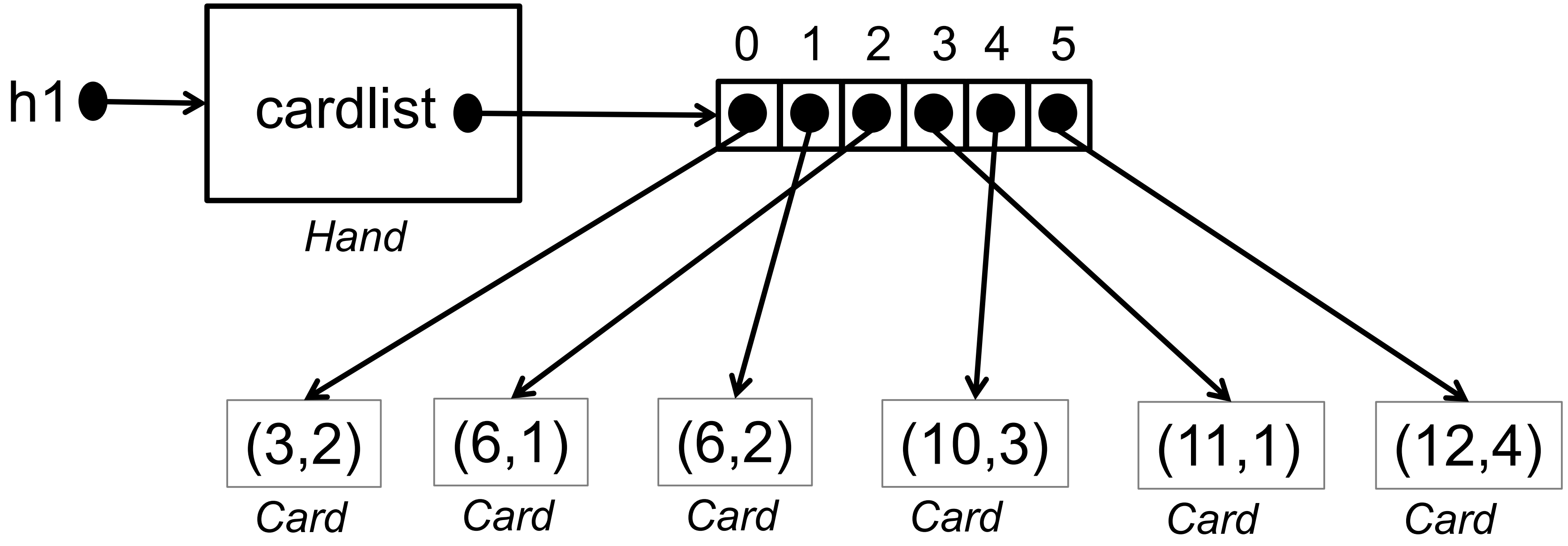


# Python Lists are *array-based* lists

A list in Python is a sequence of references to objects. The references in the list are maintained internally in an array.

```
mylist = [3,4,7,6,3,4,1,8,3,9,7,8]
```


0 1 2 3 4 5 6 7 8 9 10 11
mylist
*array contains **references**, so all of same type and size*
*objects can be of mixed type or size, and are stored anywhere Python wants*
0 1 2 3 4 5 6 7 8 9 10
int int int int int int int int int int int


h1
cardlist
Hand
0
1
2
3
4
5
(3,2)
Card
(6,1)
Card
(6,2)
Card
(10,3)
Card
(11,1)
Card
(12,4)
Card

## Tuples

A tuple is a sequence of references to objects. The length and content of a tuple cannot change (i.e. it is *immutable*).

Implementation is similar to Lists, but is able to take advantage of the fact that they are immutable.

# 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 on the space required for the basic objects.

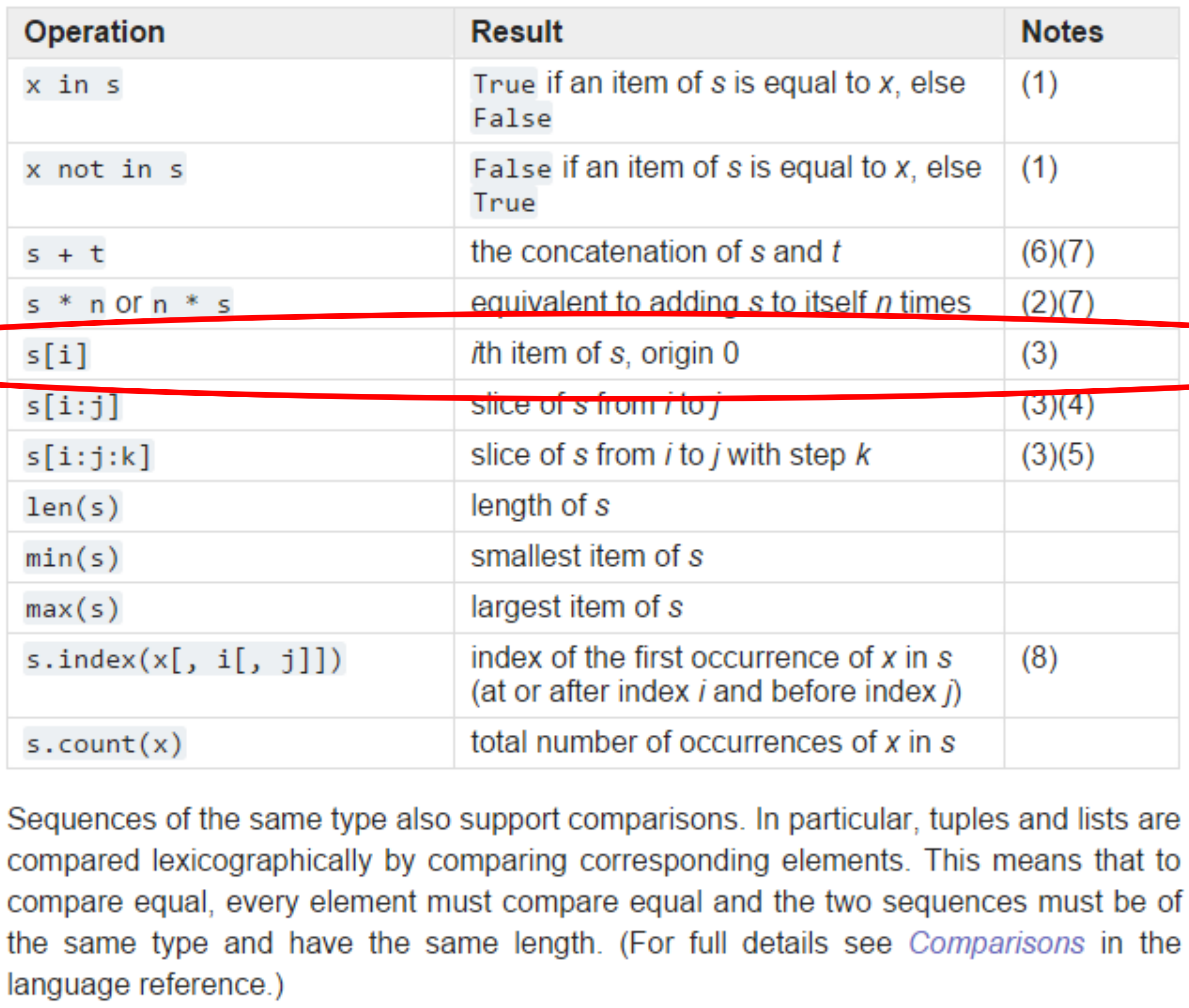
The objects will occupy space somewhere else in memory

a list of integers

a list of tuples each of 10 integers

```
import sys
def list_size():
    mylist = []
    biglist = []
    for i in range(1000):
        mylist.append(1)
        biglist.append((1,2,3,4,5,6,7,8,9,10))
    length = len(mylist)
    size = sys.getsizeof(mylist)
    bigsize = sys.getsizeof(biglist)
    print('List of', length, 'ints takes size (bytes)', size,
            'and list of 1000 tuples of 10 ints takes', bigsize)
```

# Sequence Operations



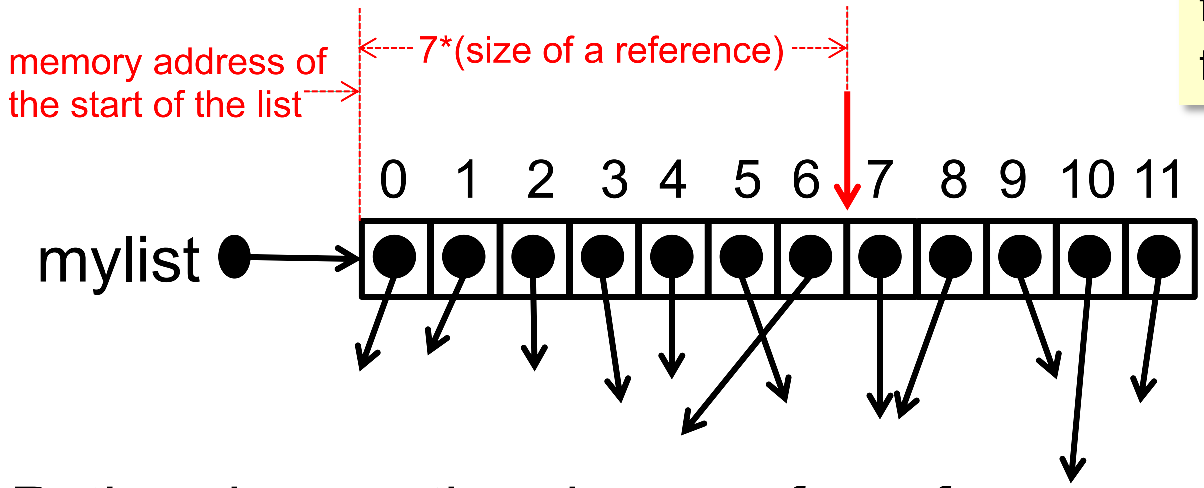
| Operation | Result | Notes |
|---|---|---|
| `x in s` | `True` if an item of *s* is equal to *x*, else `False` | (1) |
| `x not in s` | `False` if an item of *s* is equal to *x*, else `True` | (1) |
| `s + t` | the concatenation of *s* and *t* | (6)(7) |
| `s * n` or `n * s` | equivalent to adding *s* to itself *n* times | (2)(7) |
| `s[i]` | *i*th item of *s*, origin 0 | (3) |
| `s[i:j]` | slice of *s* from *i* to *j* | (3)(4) |
| `s[i:j:k]` | slice of *s* from *i* to *j* with step *k* | (3)(5) |
| `len(s)` | length of *s* | |
| `min(s)` | smallest item of *s* | |
| `max(s)` | largest item of *s* | |
| `s.index(x[, i[, j]])` | index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*) | (8) |
| `s.count(x)` | total number of occurrences of *x* in *s* | |

Sequences of the same type also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. This means that to compare equal, every element must compare equal and the two sequences must be of the same type and have the same length. (For full details see *Comparisons* in the language reference.)

# Accessing element by index

```
x = mylist[7]
```

Accessing a list element by index takes constant time, O(1), regardless of the length of the list, or the size of the index

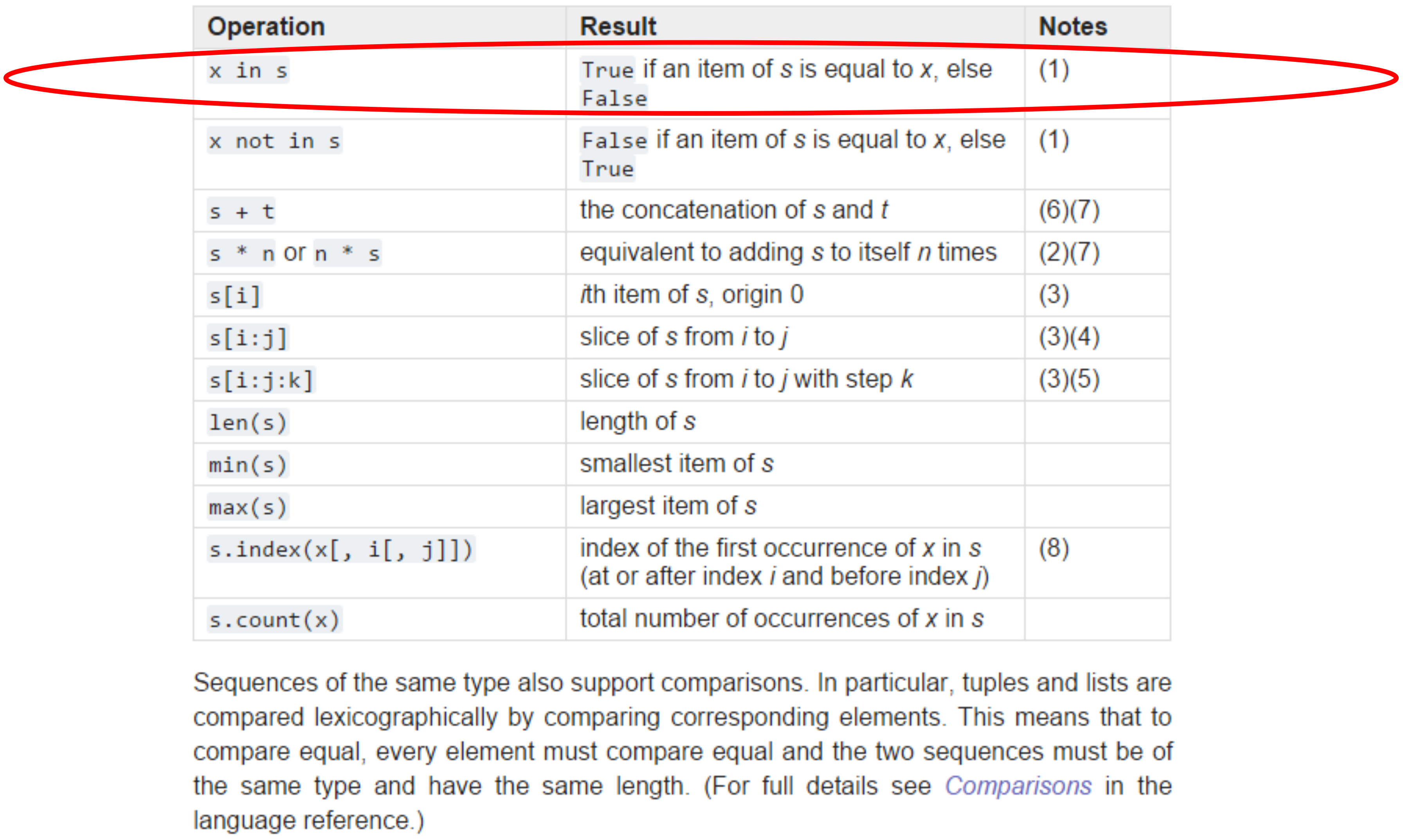


Python knows the size, *s*, of a reference, and knows the location in memory of the start of the list, *mylist*.

Compute the location of the reference by *mylist* + *s*index*

Go straight to that location to read the reference.

# Sequence Operations



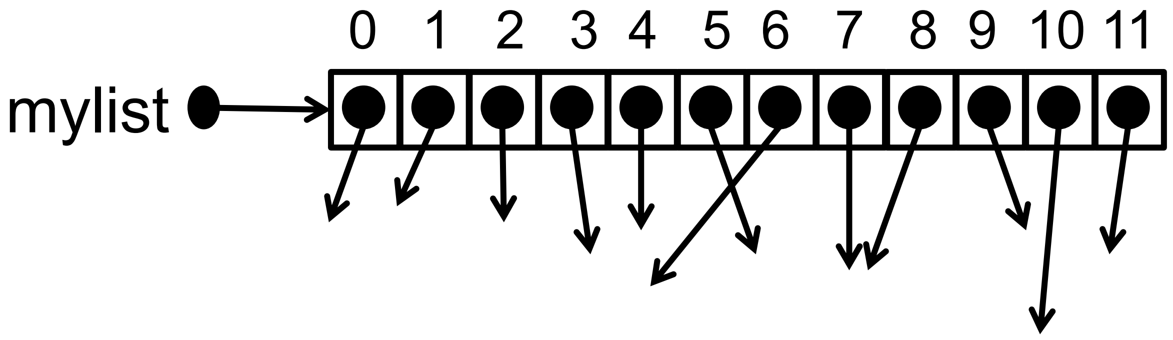
| Operation | Result | Notes |
|---|---|---|
| `x in s` | `True` if an item of *s* is equal to *x*, else `False` | (1) |
| `x not in s` | `False` if an item of *s* is equal to *x*, else `True` | (1) |
| `s + t` | the concatenation of *s* and *t* | (6)(7) |
| `s * n` or `n * s` | equivalent to adding *s* to itself *n* times | (2)(7) |
| `s[i]` | *i*th item of *s*, origin 0 | (3) |
| `s[i:j]` | slice of *s* from *i* to *j* | (3)(4) |
| `s[i:j:k]` | slice of *s* from *i* to *j* with step *k* | (3)(5) |
| `len(s)` | length of *s* | |
| `min(s)` | smallest item of *s* | |
| `max(s)` | largest item of *s* | |
| `s.index(x[, i[, j]])` | index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*) | (8) |
| `s.count(x)` | total number of occurrences of *x* in *s* | |

Sequences of the same type also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. This means that to compare equal, every element must compare equal and the two sequences must be of the same type and have the same length. (For full details see *Comparisons* in the language reference.)

# Searching for an element

Finding an element in a list takes time O(n), where n is the length of the list

```
if 4 in mylist:
```

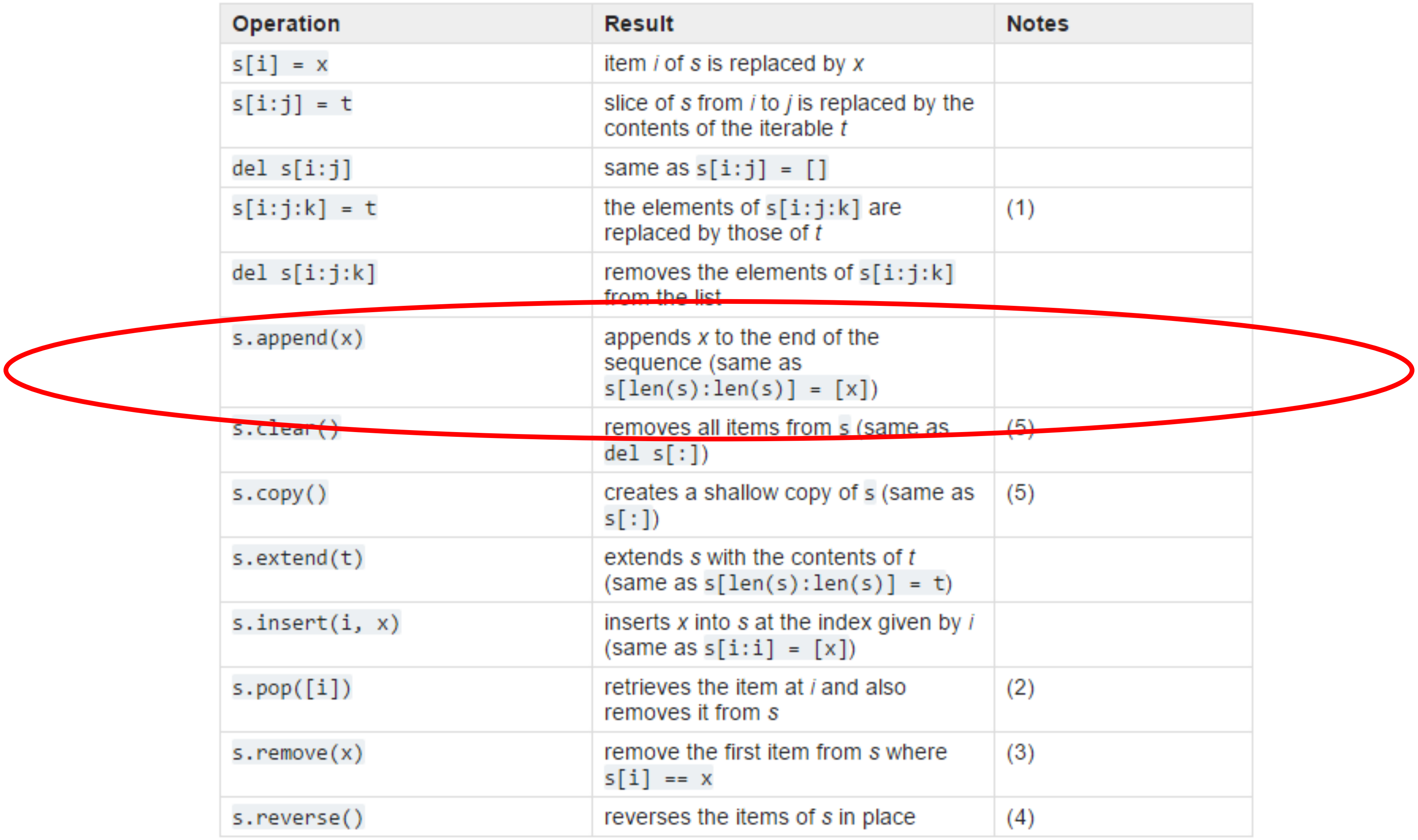


```
if Card(3,3) in mylist:
```

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

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

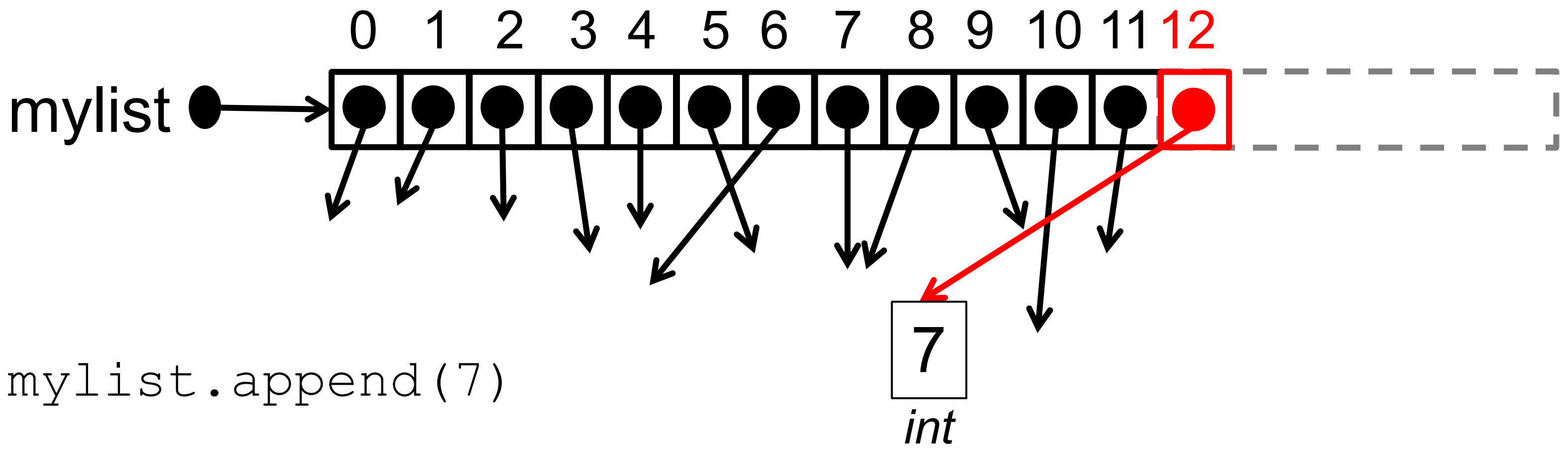
# ... and for *mutable* sequences



| Operation | Result | Notes |
|---|---|---|
| `s[i] = x` | item *i* of *s* is replaced by *x* | |
| `s[i:j] = t` | slice of *s* from *i* to *j* is replaced by the contents of the iterable *t* | |
| `del s[i:j]` | same as `s[i:j] = []` | |
| `s[i:j:k] = t` | the elements of `s[i:j:k]` are replaced by those of *t* | (1) |
| `del s[i:j:k]` | removes the elements of `s[i:j:k]` from the list | |
| `s.append(x)` | appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`) | |
| `s.clear()` | removes all items from `s` (same as `del s[:]`) | (5) |
| `s.copy()` | creates a shallow copy of `s` (same as `s[:]`) | (5) |
| `s.extend(t)` | extends *s* with the contents of *t* (same as `s[len(s):len(s)] = t`) | |
| `s.insert(i, x)` | inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`) | |
| `s.pop([i])` | retrieves the item at *i* and also removes it from *s* | (2) |
| `s.remove(x)` | remove the first item from *s* where `s[i] == x` | (3) |
| `s.reverse()` | reverses the items of *s* in place | (4) |

# Increasing list size (1)

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

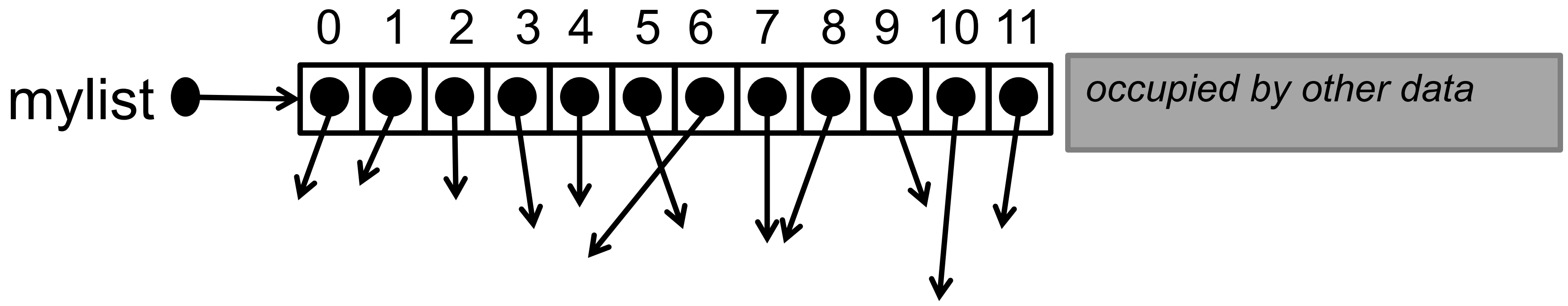


```
mylist.append(7)
```

If we already have space reserved at the end of the list, use it ...

## Increasing list size (2)

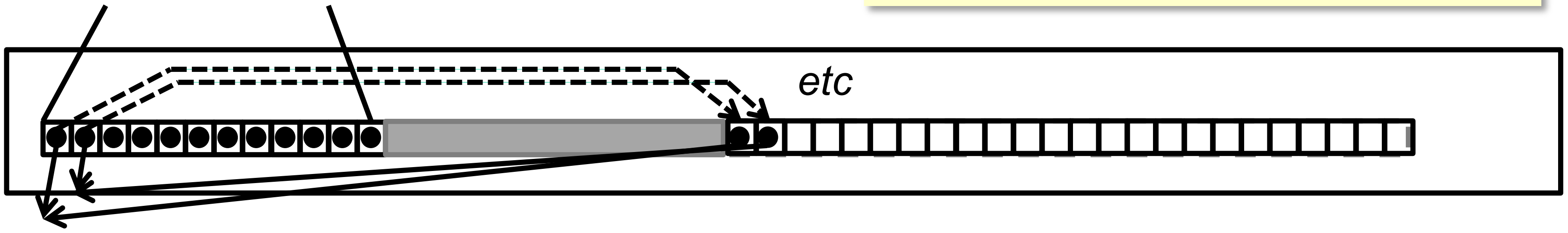
What happens if there is no reserved space?



```
mylist.append(7)
```

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

*and now free up all of this*



# Naive append

If python added one new cell into the list each time we append

- each append would require a full copy of the current list

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

For a list that may grow to size *n*, using naive append takes O($n^2$) to build the whole list

1 copy op +
2 copy ops +
3 copy ops +
4 copy ops +
...
999 copy ops
= 0.5*999*1000
= 499500 copy ops

# Python's dynamic lists

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

```
List of length 0 takes size (bytes) 36
List of length 1 takes size (bytes) 52
List of length 2 takes size (bytes) 52
...
...
List of length 87 takes size (bytes) 388
List of length 88 takes size (bytes) 388
List of length 89 takes size (bytes) 460
List of length 90 takes size (bytes) 460
..
List of length 106 takes size (bytes) 460
List of length 107 takes size (bytes) 540
```

# Complexity of list doubling

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

| | |
|---|---|
| To create a list that reaches size *n*, this will take | |
| 1 assignment + | up to 1 element |
| 1 copy op + 1 assignment + | up to 2 elements |
| 2 copy ops + 2 assignments + | up to 4 elements |
| 4 copy ops + 4 assignments + | up to 8 elements |
| ... | … |
| n/2 copy ops + n/2 assignments | up to *n* elements |

1+2+4+...+(n/2) copies + 1+1+2+4+...+(n/2) assignments

$2^0 + 2^1 + 2^2 + \dots 2^{\log(n)-1}$ copies $+ 2^0 + 2^1 + 2^2 + \dots + 2^{\log(n)-1}$ assignments

$\boxed{?}$

Note: $n = 2^{\log(n)}$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

This result is needed in the analysis of the runtime of various algorithms.

<u>Proof</u>

# Complexity of list doubling

Policy: each time we must grow the list, *double* its size. Suppose we are unlucky, 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 op + 1 assignment +
2 copy ops + 2 assignments +
4 copy ops + 4 assignments +
...
n/2 copy ops + n/2 assignments

1+2+4+...+(n/2) copies + 1+1+2+4+...+(n/2) assignments

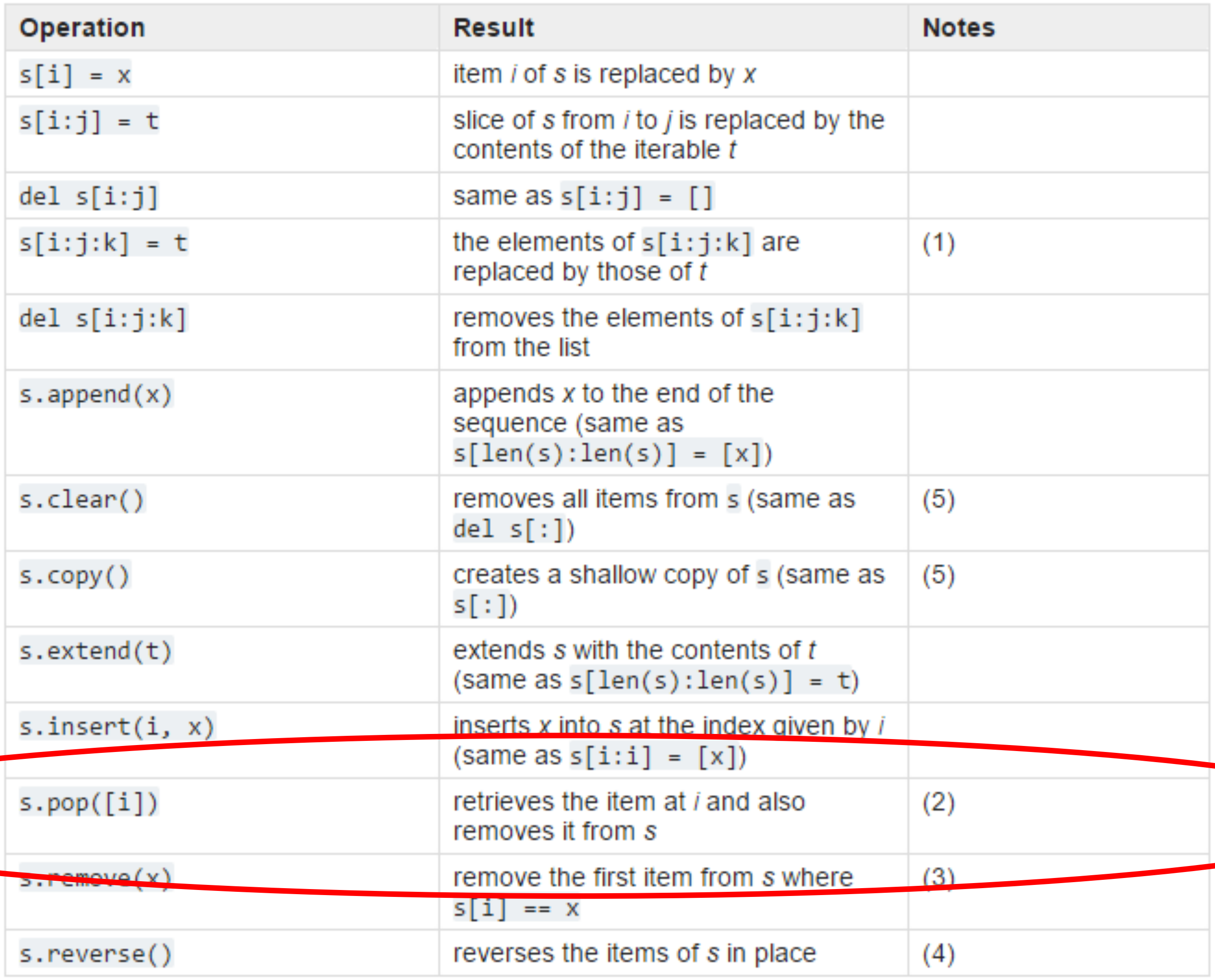
$2^0 + 2^1 + 2^2 + \ldots 2^{\log(n)-1}$ copies + $2^0 + 2^1 + 2^2 + \ldots + 2^{\log(n)-1}$ assignments

$2^{\log(n)}-1$ copies + $2^{\log(n)}-1$ assignments
n-1 copies + n-1 assignments, which is O(n)

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 then *O(1)*

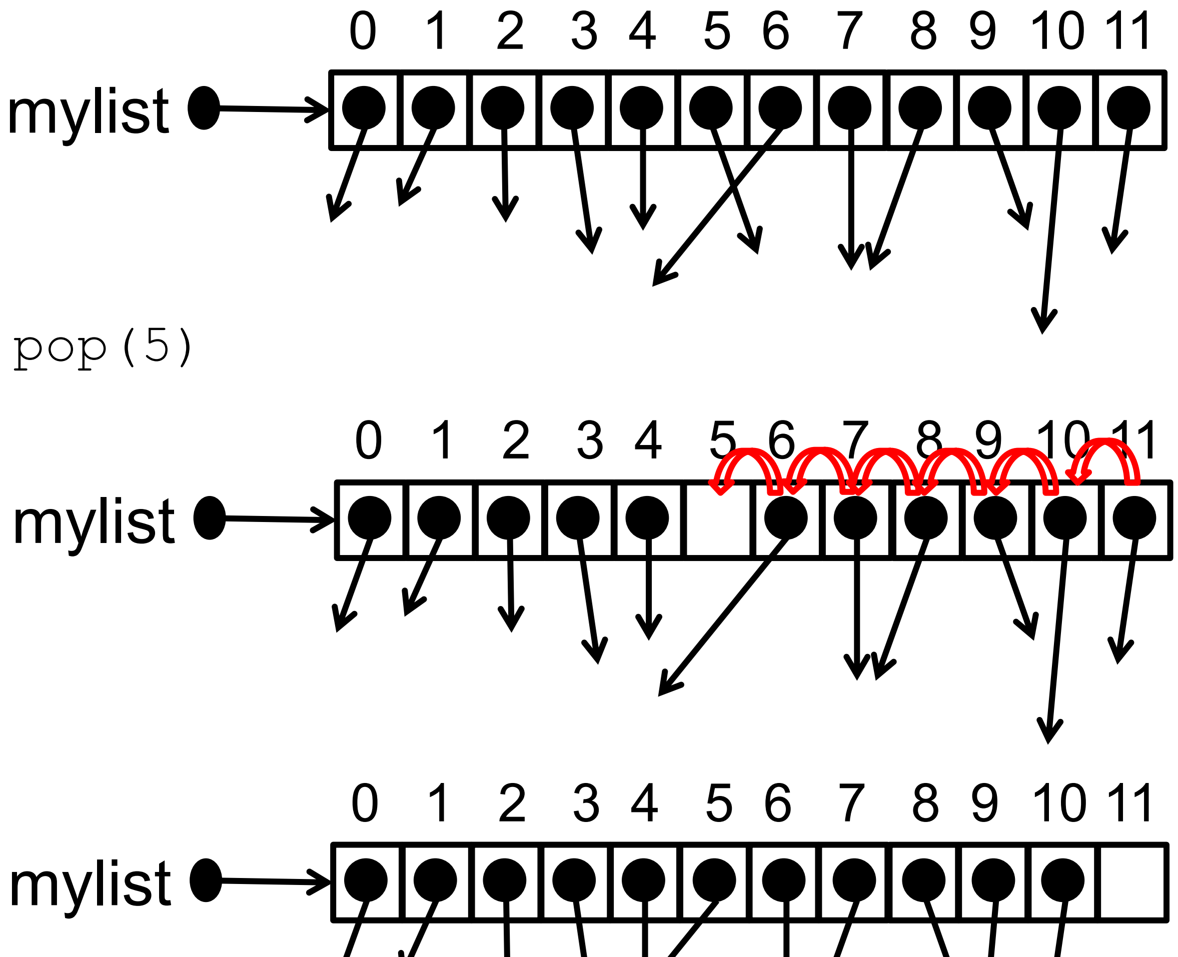
# ... and for *mutable* sequences



| Operation | Result | Notes |
|---|---|---|
| `s[i] = x` | item *i* of *s* is replaced by *x* | |
| `s[i:j] = t` | slice of *s* from *i* to *j* is replaced by the contents of the iterable *t* | |
| `del s[i:j]` | same as `s[i:j] = []` | |
| `s[i:j:k] = t` | the elements of `s[i:j:k]` are replaced by those of *t* | (1) |
| `del s[i:j:k]` | removes the elements of `s[i:j:k]` from the list | |
| `s.append(x)` | appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`) | |
| `s.clear()` | removes all items from `s` (same as `del s[:]`) | (5) |
| `s.copy()` | creates a shallow copy of `s` (same as `s[:]`) | (5) |
| `s.extend(t)` | extends *s* with the contents of *t* (same as `s[len(s):len(s)] = t`) | |
| `s.insert(i, x)` | inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`) | |
| `s.pop([i])` | retrieves the item at *i* and also removes it from *s* | (2) |
| `s.remove(x)` | remove the first item from *s* where `s[i] == x` | (3) |
| `s.reverse()` | reverses the items of *s* in place | (4) |

# Cost of popping an element

Python's lists have no spaces – when we pop an element, the list must close up, pushing space to the end.

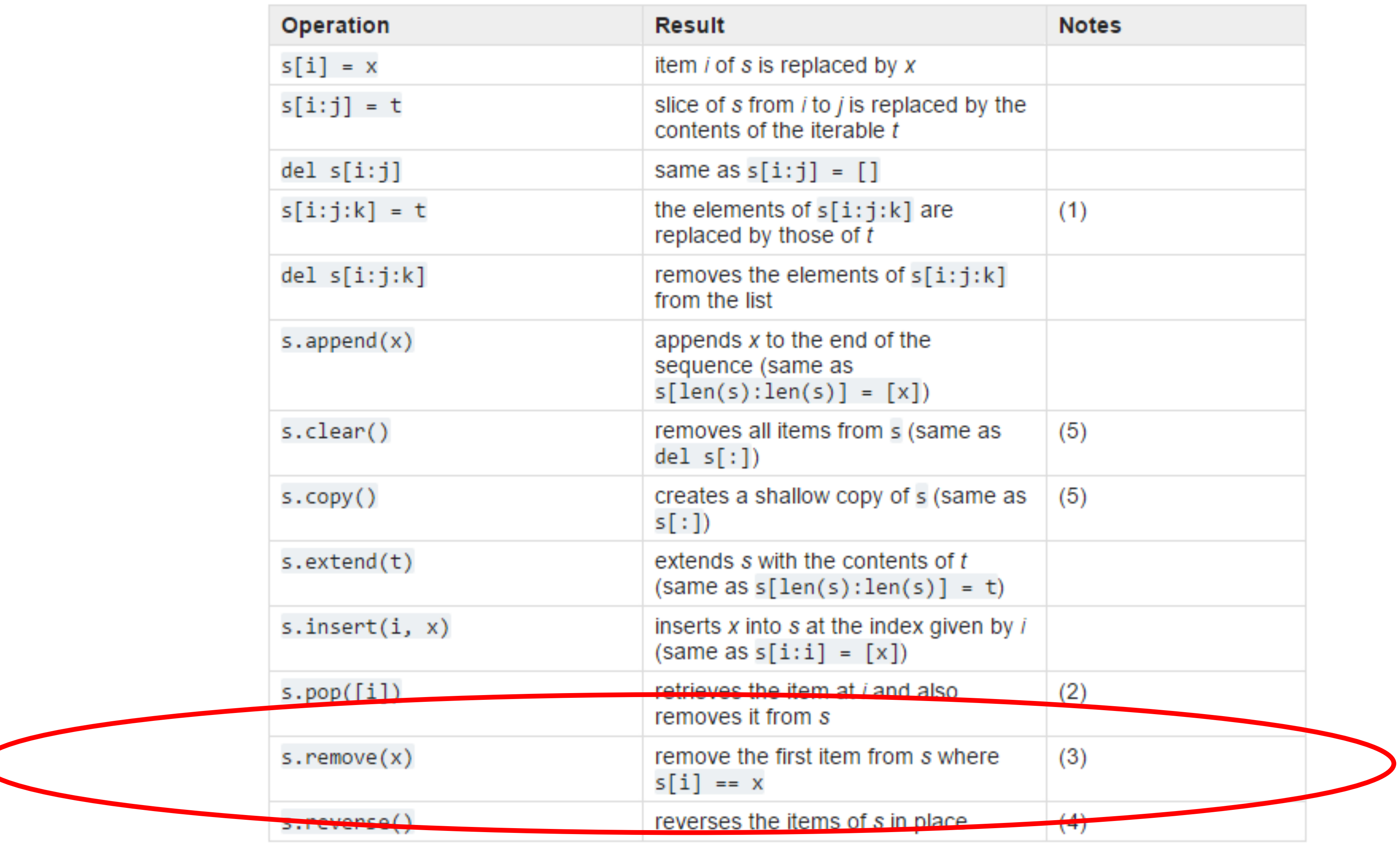


for a list of length *n*, pop(i) takes *(n-1-i)* copies.

pop(0) takes *n-1* copies

pop is *O(n)*

# ... and for *mutable* sequences



| Operation | Result | Notes |
|---|---|---|
| `s[i] = x` | item *i* of *s* is replaced by *x* | |
| `s[i:j] = t` | slice of *s* from *i* to *j* is replaced by the contents of the iterable *t* | |
| `del s[i:j]` | same as `s[i:j] = []` | |
| `s[i:j:k] = t` | the elements of `s[i:j:k]` are replaced by those of *t* | (1) |
| `del s[i:j:k]` | removes the elements of `s[i:j:k]` from the list | |
| `s.append(x)` | appends *x* to the end of the sequence (same as `s[len(s):len(s)] = [x]`) | |
| `s.clear()` | removes all items from `s` (same as `del s[:]`) | (5) |
| `s.copy()` | creates a shallow copy of `s` (same as `s[:]`) | (5) |
| `s.extend(t)` | extends *s* with the contents of *t* (same as `s[len(s):len(s)] = t`) | |
| `s.insert(i, x)` | inserts *x* into *s* at the index given by *i* (same as `s[i:i] = [x]`) | |
| `s.pop([i])` | retrieves the item at *i* and also removes it from *s* | (2) |
| `s.remove(x)` | remove the first item from *s* where `s[i] == x` | (3) |
| `s.reverse()` | reverses the items of *s* in place | (4) |

## The remove(...) method

`mylist.remove(x)` will remove the first occurrence of x that it finds in the list.

Exercise: what is the worst case time complexity of remove? What is the complexity for an arbitrary element?

Exercise 2: remove(...) only removes the first instance it finds in the list, and leaves any duplicates in place.
Write an efficient method to remove all instances

## Deleting an element: space

del, pop and remove 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.

## Next lecture ...

Defining commonly used access patterns for sequence data:
Stacks