Sequences in Python: Lists, Tuples, and strings

Pai H. Chou
National Tsing Hua University

Outline

- Strings, lists, tuples
- Operators on sequences
  - comparison (lexicographical order)
  - membership, concatenation, repetition, slice
- Methods and functions
  - Mutation methods (lists) vs. non-mutation methods (sequences)
  - built-in functions on sequences
  - Stacks and Queues
  - list comprehension
- shallow vs. deep copy

Sequences in Python

- Refer to
    - strings, lists, tuples
- Can be operated on by

• indexing	`s[i]`
• slicing	`s[i:j]`
• replicating n times	`s * n`
• concatenating with another sequence	`s1 + s2`
• testing membership	`obj in s`
• Comparing value for equality	`s1 == s2`

indexing operator

- Two ways of indexing a sequence *L*
  - from beginning: L[0], L[1], ... L[n-1]
  - from the end: L[-1], L[-2], L[-3], ... L[-n]
- n = `len(`*L*`)` # length of sequence

```
>>> L = tuple('ABCDE')
>>> L
('A', 'B', 'C', 'D', 'E')
>>> L[2]
'C'
>>> L[-1], L[-2], L[-3], L[-4], L[-5]
('E', 'D', 'C', 'B', 'A')
```

- same kind of index works for lists and tuples

slicing operator

- *L*[*start* : *limit*]
  - from start up to but not including limit

```
>>> L = 'ABCDE'
>>> L[1:4]
'BCD'
>>> L[-5:-2]
'ABC'
```

- Can leave out either start, limit, or both

```
>>> L[:2]
'AB'
>>> L[-3:]
'CDE'
>>> L[:]
'ABCDE'
```

Slicing with step

- *L*[*start* : *limit* : *step*]
  - default step is +1
  - step can be 2, 3, ... or -1, -2, ...

```
>>> import string
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.ascii_letters[::-1]
'ZYXWVUTSRQPONMLKJIHGFEDCBAzyxwvutsrqponmlkjihgfedcba'
>>> string.ascii_letters[::2]
'acegikmoqsuwyACEGIKMOQSUWY'
>>> string.ascii_letters[26:52:3]
'ADGJMPSVY'
>>> string.ascii_letters[51:25:-3]
'ZWTQNKHEB'
```

Slicing in assignment

- Can replace a slice with another slice
  - can be different sizes!

```
>>> L = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> L[1:4] = ['1', '2']  # replace ['b', 'c', 'd']
>>> L
['a', '1', '2', 'e', 'f', 'g']
>>> L[0:0] = ['x', 'y', 'z']   # insert at beginning
>>> L
['x', 'y', 'z', 'a', '1', '2', 'e', 'f', 'g']
>>> L[1:3] = []    # delete slice by assigning slice to empty list
>>> L
['x', 'a', '1', '2', 'e', 'f', 'g']
```

Conversion to list or tuple

- `list(s)`: converts `s` into a list
- `tuple(s)`: converts *s* into a tuple
  - `s` must be a sequence (or "iterable")

```
>>> list('abcde')
['a', 'b', 'c', 'd', 'e']
>>> tuple('abcde')
('a', 'b', 'c', 'd', 'e')
>>> list((2, 3, 1))
[2, 3, 1]
```

Operators on sequences

- comparison (lexicographical order)
- membership (`in`, `not in`)
- concatenation `+`
- repetition `*`

Comparison operators

- >, >=, <, <=, ==, !=
  - Compare prefix starting from [0]
  - Continue until either one runs out of elements, or if their difference can be resolved

```
>>> 'Apple' < 'apple'  # uppercase < lower case
True
>>> 'apple' >= 'applesauce'
False
>>> (1, 2, 3) == (1, 3) # tuples compared lexicographically
False
>>> (1, 2, 3) < (1, 3) # tuples compared lexicographically
True
```

Membership test: `in`, `not in`

- whether a substring is in a string, or a value is in a list or tuple.

```
>>> 's' in 'school'
True
>>> 'k' in 'school'
False
>>> 'sch' in 'school'
True
>>> 1 in (1, 2, 3)
True
>>> (1, 2) in (1, 2, 3)
False
>>> (1, 2) in (1, (1, 2), 3)
True
```

Concatenation with +

- Make a new sequence that concatenates two
    - must be of same type, or else error

```
>>> 'counter' + 'clockwise'
'counterclockwise'
>>> [1, 2] + [3, 4, 5]
[1, 2, 3, 4, 5]
>>> ('a', 'b', 'c') + ('d', 'e')
('a', 'b', 'c', 'd', 'e')
>>> ['counter'] + 'clockwise'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list
```

Repetition with *

- *s* * *n*: concatenates *n* copies of *s*

```
>>> 'we' + 'e' * 10
'weeeeeeeeeee'
>>> 'do' * 5
'dododododo'
>>> x = (1, 2)
>>> y = (3, x)    # y is (3, x) which is (3, (1, 2))
>>> y
(3, (1, 2))
>>> y * 2
(3, (1, 2), 3, (1, 2))  # (1, 2) appears twice, but not 2 copies
```

Methods of sequence classes

- Mutation methods - list only

`append(e)`	add e to end of list
`extend(L)`	add L[:] to end of list
`pop()`	remove last element
`insert(p,e)`	insert *e* at position *p*
`reverse()`	reverse items in list
`sort()`	sort elements in list
`remove(e)`	remove 1st occur. of *e*
`clear()`	remove all items in list

- all sequences (str, tuple, list)

`index(e)`	index of 1st occur. of *e*
`count(e)`	#times e occurs in list

Changing size of list

- What you can do with a list:
  - `list.append(`*e*`)` adds element *e* to end of list
  - `list.pop()` "pops" element from end of list
  - **`del`**`(`*item*`)` deletes *item* (from list)

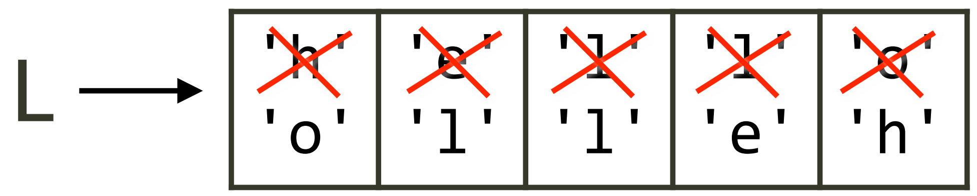
```
>>> L = ['Sun', 'Mon', 'Tue']
>>> L.append('Wed')          # add to the end
>>> L
['Sun', 'Mon', 'Tue', 'Wed']
>>> L.pop()
'Wed'
>>> L
['Sun', 'Mon', 'Tue']
>>> del(L[1])
>>> L
['Sun', 'Tue']
```

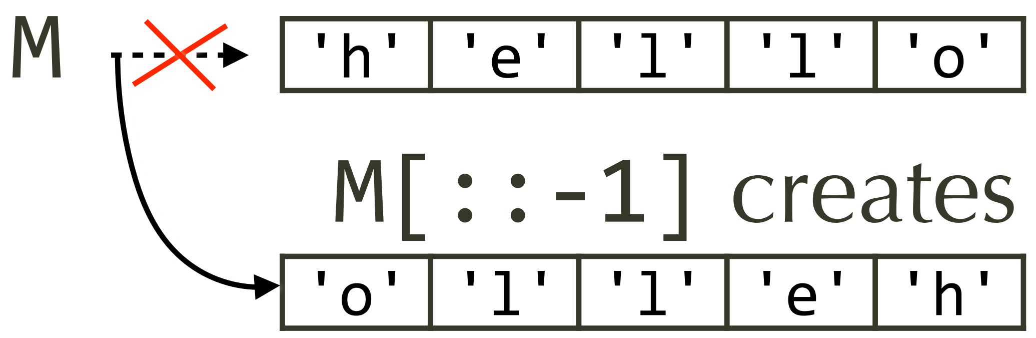
Difference between mutation and create-and-reassign

- `L.reverse()`
- `M = M[::-1]`

```
>>> L = ['h', 'e', 'l', 'l', 'o']
>>> L.reverse()
>>> L
['o', 'l', 'l', 'e', 'h']
```

```
>>> M = ['h', 'e', 'l', 'l', 'o']
>>> M = M[::-1]
>>> M
['o', 'l', 'l', 'e', 'h']
```





`L.reverse()` reorders the elements in the same list (does not create a new list)

`M` refers to a newly created list (old list may become "garbage" if no longer accessible)

Example list methods

```
>>> L = list(range(4))
>>> L
[0, 1, 2, 3]
>>> L.append('A')
>>> L
[0, 1, 2, 3, 'A']
>>> L.pop()
'A'
>>> L
[0, 1, 2, 3]
>>> L.insert(3, 'z')
>>> L
[0, 1, 2, 'z', 3]
>>> L.extend(['y', 'z'])
>>> L
[0, 1, 2, 'z', 3, 'y', 'z']
```

```
>>> L.count('z')
2
>>> L.index('z')
3
>>> L.remove('z')
>>> L
[0, 1, 2, 3, 'y', 'z']
>>> L.reverse()
>>> L
['z', 'y', 3, 2, 1, 0]
>>> L.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between
instances of 'int' and 'str'
```

Stacks

- Last-in, First-out (LIFO) data structure
  - push: add element to top of stack (list **append**)
  - pop: remove element from top of stack (list **pop**)

```
>>> L = ['a', 'b', 'c']
>>> L.append(3)  # push
>>> L
['a', 'b', 'c', 3]
>>> L.pop()
3
>>> L.pop()
'c'
>>> L
['a', 'b']
```

Queues

- First-in, First-out (FIFO) data structure
  - enqueue: add element to tail of queue (`append`)
  - dequeue: remove element from head of queue (`pop(0)`)

```
>>> L = ['a', 'b', 'c']
>>> L.append(3)  # enqueue
>>> L
['a', 'b', 'c', 3]
>>> L.pop(0)     # dequeue
'a'
>>> L.pop(0)     # dequeue
'b'
>>> L
['c', 3]
```

built-in functions

- min(L), max(L), sum(L)
- any(L), all(L)
- sorted(L)
- reversed(L)

min(), max(), sum()

- min() or max() element in S
  - S can be a string, tuple, or list

```
>>> s = 'ABCDE'
>>> max(s), min(s)
>>> ('E', 'A')
>>> max(2, 5, 1)    # also works as individual arguments
5
```

- sum(): numeric total of items in list
  - must be number! cannot be string

```
>>> s = [1, 5, 3, 2, 8]
>>> sum(s)
19
```

`any()`, `all()`

- any: returns True if any element is True
- all: returns True if all elements are True
  - Recall: zero, empty container => False; nonzero, nonempty => True

```
>>> L = ['', 'apple', 'oranges', 'banana']
>>> any(L)
True
>>> all(L)
False
>>> M = [0, '', 0.0, [], ()]
>>> any(M)
False
>>> all(M)
False
```

sorted()

- makes a copy of the same type of data structure but with elements in sorted order
    - does not modify sequence!
    - works for str, list, tuple

```
>>> s = [1, 5, 3, 2, 8]
>>> sorted(s)
[1, 2, 3, 5, 8]
```

- However, items must be comparable types

```
>>> sorted([1, (), '', []])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'tuple' and
'int'
```

`reversed()`

- makes an iterator with elements in reverse order
    - does not modify the sequence!

```
>>> s = [1, 5, 3, 2, 8]
>>> reversed(s)
<reversed object at 0x10a55bf98>
```

- to see content of iterator: convert to list

```
>>> list(reversed(s))
[8, 2, 3, 5, 1]
```

List: mutate vs. create+abandon

- Two ways of modifying a list variable
  - Mutation: by method call, function call, operator, including incremental assignment operator
  - Create + abandon, no mutation
- To modify a variable to immutable
  - create new + abandon old
  - less efficient than mutation

Options for Mutable vs. Immutable Data Structures

• Mutation (list only)	• Non-mutation (all)
• L.sort()	• S = sorted(S)
• L.reverse()	• S = list(reversed(S)) S = S[::-1]
• L.extend([1, 2, 3]) L += [1, 2, 3]	• S = S + [1, 2, 3]
• del(L[1])	• S = S[:1] + S[2:]
• L.pop()	• S = S[:-1]

list comprehension

- make a list using loops and expressions
  - [*expression* **for** *loopVar* **in** *iteration*]

```
>>> [chr(65+i) for i in range(5)] # 65 is ASCII for 'A'
['A', 'B', 'C', 'D', 'E']
>>> [2**i for i in range(1, 11)] # powers of 2 up to 2^10
[2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
>>> [(chr(i), i) for i in range(65, 70)] # tuples of (char, code)
[('A', 65), ('B', 66), ('C', 67), ('D', 68), ('E', 69)]
```

list comprehension with a condition

- add if condition after in
  - [*expression* **for** *loopVar* **in** *iteration* **if** *cond*]

```
>>> [chr(i) for i in range(65, 65+26)]    # all uppercase letters
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
>>> [chr(i) for i in range(65, 65+26) \
...  if chr(i) not in ['A','E','I','O','U']] # non-vowel subset
['B', 'C', 'D', 'F', 'G', 'H', 'J', 'K', 'L', 'M', 'N', 'P', 'Q',
'R', 'S', 'T', 'V', 'W', 'X', 'Y', 'Z']
>>> [i*(i+1) for i in range(11)]
[0, 2, 6, 12, 20, 30, 42, 56, 72, 90, 110]
>>> [i*(i+1) for i in range(1, 11) if i*(i+1)%3==0]
[0, 6, 12, 30, 42, 72, 90]            # filter for multiples of 3
```

multi-dimensional list comprehension

- can have multiple levels of loops
  - [*expr.* **for** *outerVar* **in** *R1* **for** *innerVar* **in** *R2*]
  - *R2* can refer to outerVar
  - can still add **if** filter

```
>>> [(i, j, i*j) for i in range(1,5) for j in range(1,5)] # mult table
[(1, 1, 1), (1, 2, 2), (1, 3, 3), (1, 4, 4), (2, 1, 2), (2, 2, 4), (2,
3, 6), (2, 4, 8), (3, 1, 3), (3, 2, 6), (3, 3, 9), (3, 4, 12), (4, 1,
4), (4, 2, 8), (4, 3, 12), (4, 4, 16)]
>>> [(i, j) for i in range(1,5) for j in range(i,5)] # upper triangle
[(1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4), (3, 3), (3,
4), (4, 4)]
>>> [(i, j) for i in range(1,5) for j in range(i,5) if i != j]
[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)] # exclude diagonals
```

Deep copy vs. shallow copy

- shallow copy:
  - make a new list that references the same elements as those in the original list
  - for elements that are mutable, any change will show up in both the original and the shallow copy
- deep copy:
  - make a full replica of the values in any nested objects
  - all elements are clones, so changes to the original element will not affect the clone, and vice versa.

Illustration of shallow copy

```
>>> X = ['A', 'B']
>>> L = [1, X, 2, 3]
>>> M = L.copy() # same as M=L[:]
>>> L   # L references list X
[1, ['A', 'B'], 2, 3]
>>> M   # M also references list X
[1, ['A', 'B'], 2, 3]
>>> X.pop()
'B'
>>> L  # L unchanged, new X shows
[1, ['A'], 2, 3]
>>> M  # also shows updated X value
[1, ['A'], 2, 3]
>>> L.extend(['y', 'z'])
>>> L  # changing L doesn't affect M
[1, ['A'], 2, 3, 'y', 'z']
>>> M  # M is unaffected.
[1, ['A'], 2, 3]
```

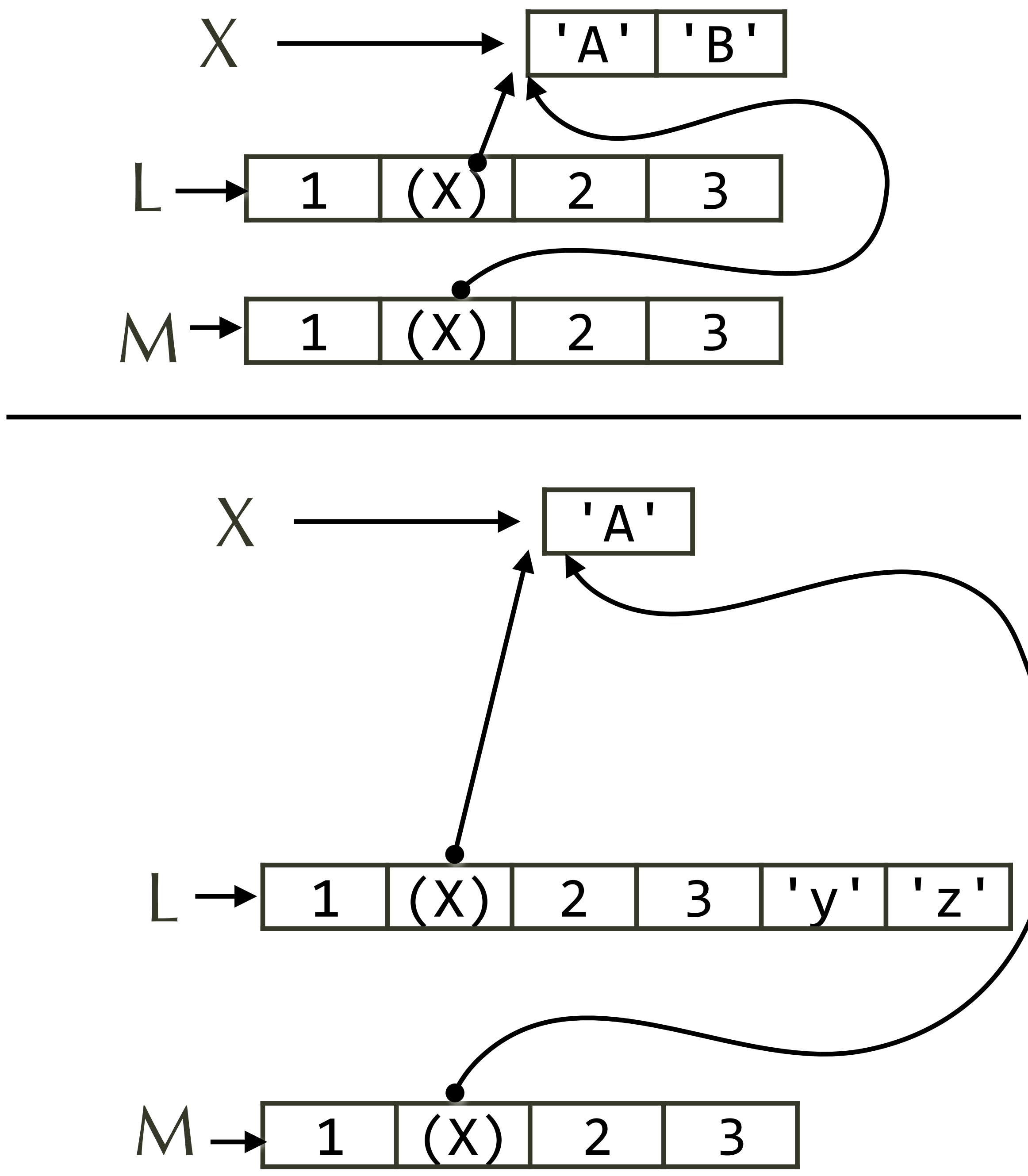
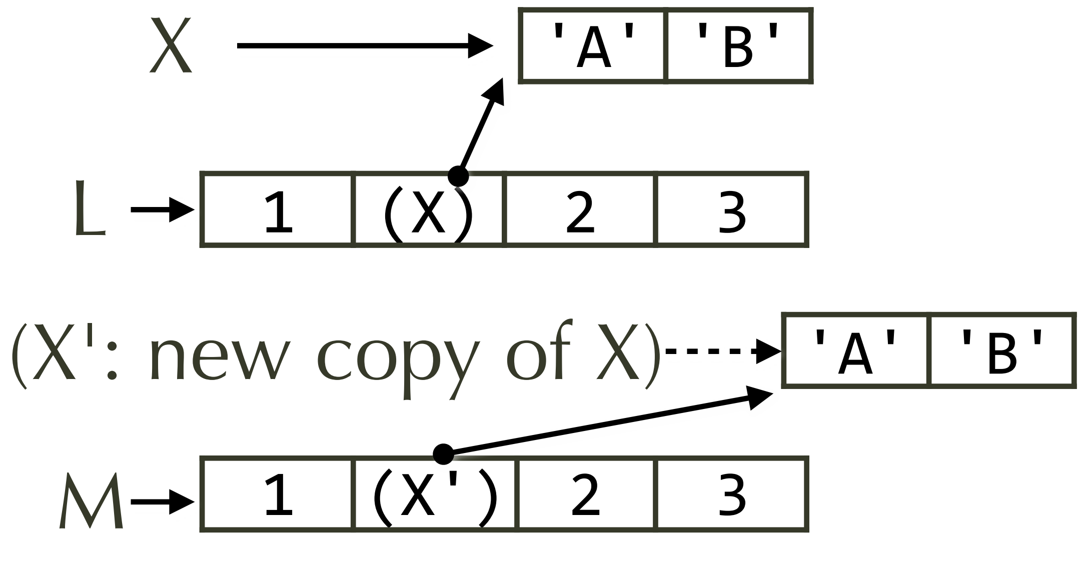
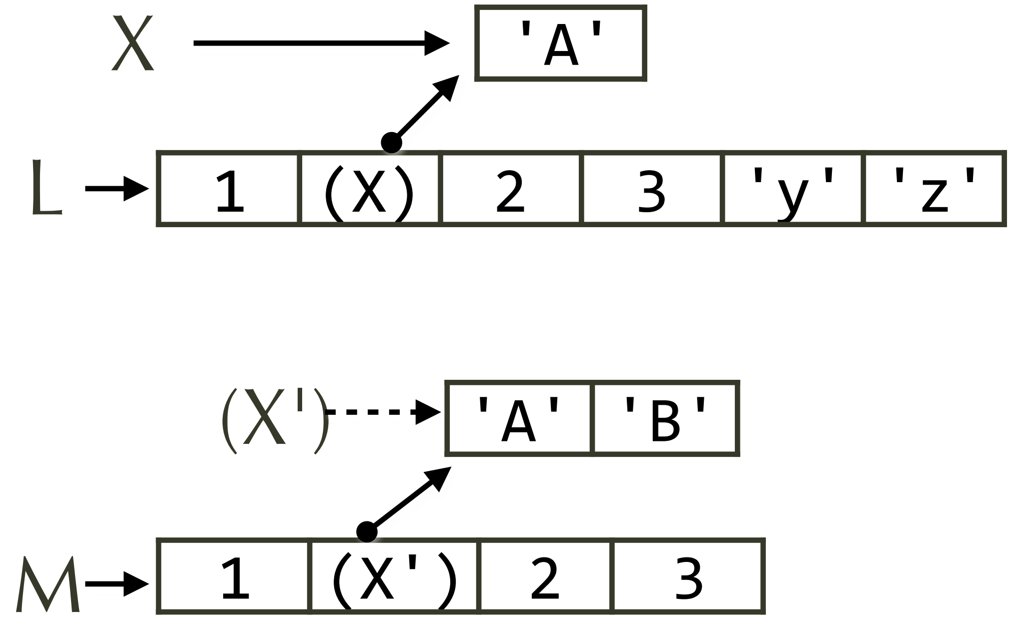


Illustration of deep copy

```
>>> import copy
>>> X = ['A', 'B']
>>> L = [1, X, 2, 3]
>>> M = copy.deepcopy(L)
>>> L   # L references list X
[1, ['A', 'B'], 2, 3]
>>> M   # M has a copy of list X
[1, ['A', 'B'], 2, 3]
>>> X.pop()
'B'
>>> L  # L unchanged, new X shows
[1, ['A'], 2, 3]
>>> M  # unaffected
[1, ['A', 'B'], 2, 3]
>>> L.extend(['y', 'z'])
>>> L  # changing L doesn't affect M
[1, ['A'], 2, 3, 'y', 'z']
>>> M  # M is unaffected.
[1, ['A', 'B'], 2, 3]
```


X
'A'
'B'
L
1
(X)
2
3
(X': new copy of X)
'A'
'B'
M
1
(X')
2
3


X
'A'
L
1
(X)
2
3
'y'
'z'
(X')
'A'
'B'
M
1
(X')
2
3