Module 04: Lists

Topics:

- Lists and their methods
- Mutating lists
- Abstract list functions

Readings: ThinkP 8, 10

Practice: Self study exercises

Consider the string method **`split`**

```
>>> name = "Harry James Potter"
>>> name.split()
['Harry', 'James', 'Potter']
>>> name.split('e')
['Harry Jam', 's Pott', 'r']
```

`split` returns a list of strings.

Lists in Python

- Like Racket lists, Python lists can store
  - any number of values
  - any types of values (even in one list)
- Creating lists:
  - Use square brackets to begin and end list, and separate elements with a comma
  - Concatenate (using +) existing lists to create a new list
- Examples:

```
num_list = [4, 5, 0]
str_list = ['a', 'b']
empty_list = []
mixed_list = ['abc', 12, True, '',-12.4]
```

Useful Information about Python Lists

- **`len(L)`** ⇨ number of items in the list **`L`**
- **`L[i]`** ⇨ item at position **`i`**
  - Called indexing the list
  - Causes an error if `i` is out of range
  - Positions: **`0 <= i < len(L)`**
  - Actual valid range: **`-len(L) <= i < len(L)`**
- "Slicing" a list

`L[i:j]` ⇨ **`[L[i],L[i+1],…,L[j-1]]`**

`L[i:j:k]` ⇨ **`[L[i],L[i+k],…,L[i+m*k]]`**, includes all the positions up to (but not including) **`j`**

Basic Template for Recursion

```
def list_template(L):
  if L == []:
     # base case action
  else:
     # … L[0] …       (first)
     # … list_template (L[1:]) …
     #                (recurse on rest)
```

Example:

Write a recursive Python function **`build_str`** that consumes a list of strings (**`los`**), and creates and returns a new string by concatenating together all the strings in **`los`**.

Aside: The following operation also solves this problem: ***`"".join(los)`***. You may use **`join`** on assignments unless told otherwise.

Other list operations

- **`in`**
  - **`x in L`** ⇨ **`True`** if **`x`** is in **`L`**, **`False`** otherwise
    - **`5 in [10,2,4,5]`** ⇨ **`True`**
    - **`"a" in ["hello", "there", "anyone"]`** ⇨ **`False`**
- **`sum`**
  - Returns the sum of all values in a list of numbers
    - **`sum([])`** ⇨ **`0`**
    - **`sum([1,2.25,0,-1])`** ⇨ **`2.25`**
    - **`sum([0,1,2,'3'])`** ⇨ **`Error`**
- **`min`**, **`max`** can consume lists as well

Other list operations

```
>>> dir(list)
[ …, 'append', 'count', 'extend',
  'index', 'insert', 'pop', 'remove',
  'reverse', 'sort']
```

- *Most* of these methods change the contents of list.
- *None* of these methods return a new list.

Using list methods

What does this do?

```
L = [1,2,3]
v = L.pop(0)
L.append(v)
```

To fully investigate how we can change the contents of a list using the list methods or other techniques, we first need to learn about ***list mutation***.

Mutation and Lists

Once a list is created

- We can change individual values in the list
- We can add values to the list
- We can remove values from the list
- Lists are ***mutable***, unlike the other values we have studied so far.

To fully understand how lists can be mutated, we need to learn the basics about how memory is managed in Python. We will use a simplified model.

Python Memory Model: Initialization using immutable values

- Consider:

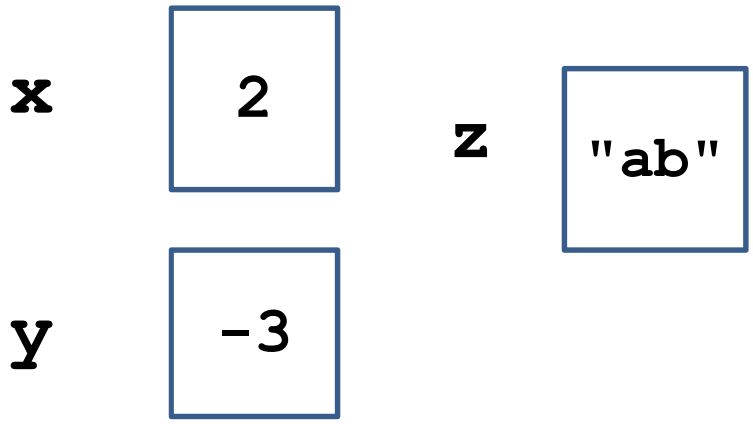
`var = expr`

where **`expr`** simplifies to **`Int`**, **`Float`**, **`Str`**, or **`Bool`**

- A box is created and labelled **`var`**
- **`expr`** is simplified to a value, and put inside the box

For example:

```
x = 2
y = 2 - 5
z = "a" + "b"
```



Python Memory Model: Changing an existing variable

- Consider:

`var = new_expr`

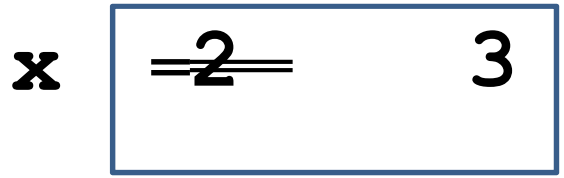
where **`var`** already has a value, and **`new_expr`** simplifies to an immutable value

- **`new_expr`** is simplified to a value, and put into the box, overwriting previous contents

For example:

`x = 2`

`x = x+1`

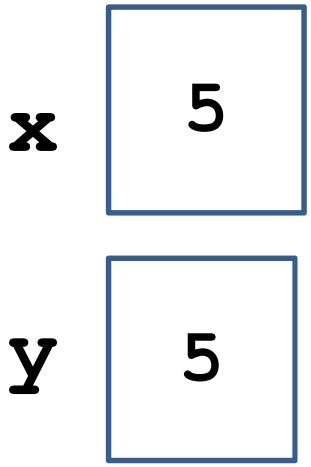


Python Memory Model: More Basics

- Suppose the value of one variable is used to initialize another variable:

```
x = 5
y = x
```

- The value in **x**'s box is copied to **y**'s box

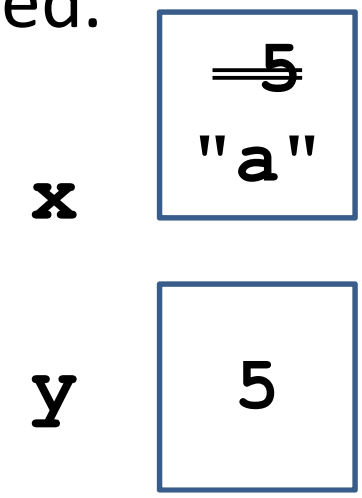


Python Memory Model: more

- If a new value is assigned to one of the variables, the variables no longer share a value

```
x = 5
y = x
x = 'a'
```

- Only one variable's value is changed. The other is unchanged.



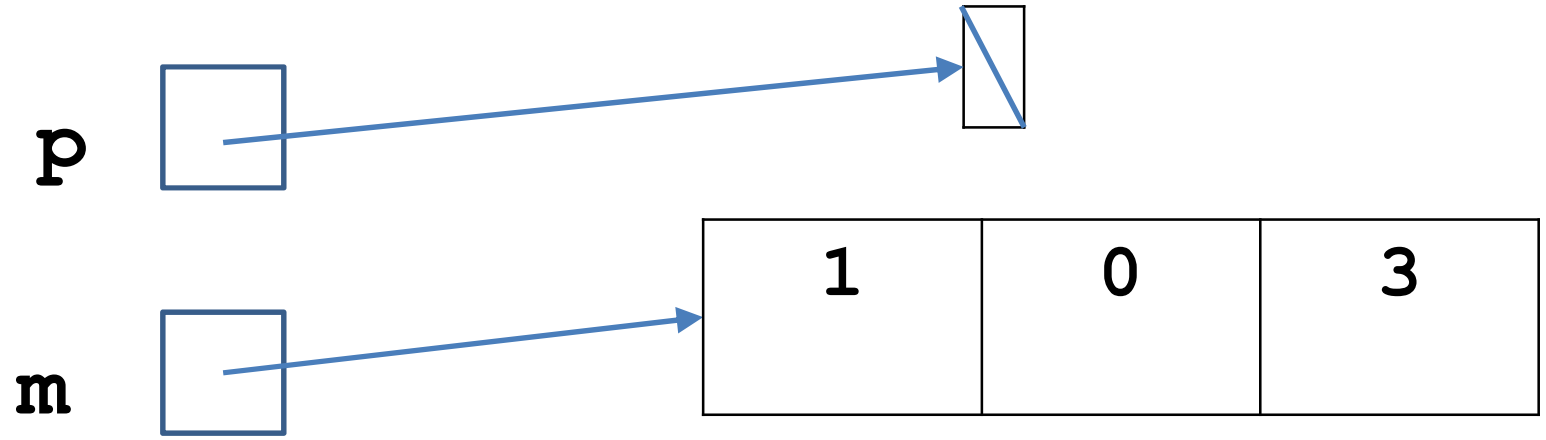
Representing lists in memory

`L = list_expr`

- Lists are comprised of multiple values, and list components can be changed, so the representation is more complicated
- The simplified value of **`list_expr`** is not put in **`L`**'s box: it gets its own space in memory which **`L`** will reference

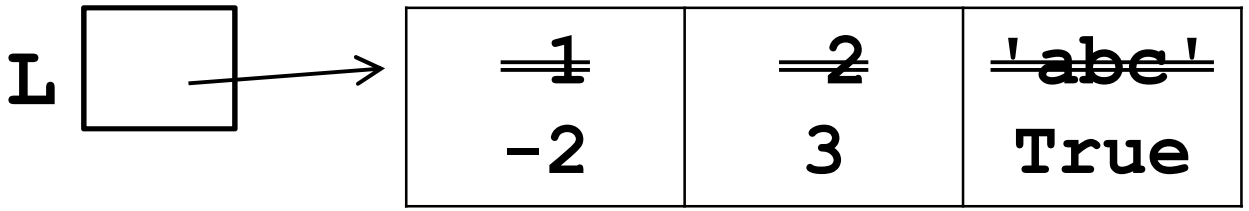
Representing lists in memory

```
p = []
m = [1,0,3]
```



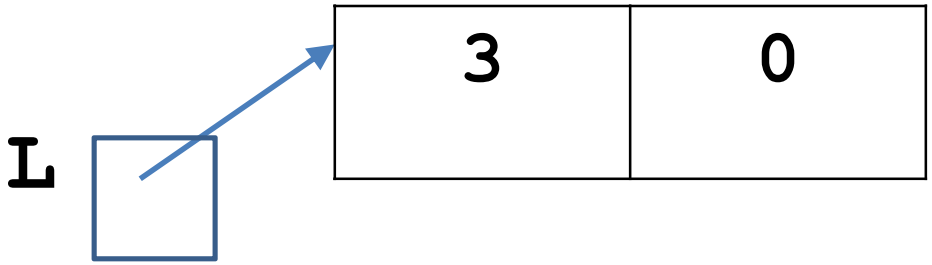
Mutation and Lists

```
L = [1,2,'abc']
L[1] = 3
L[0] = L[0]-L[1]
L[2] = True
```



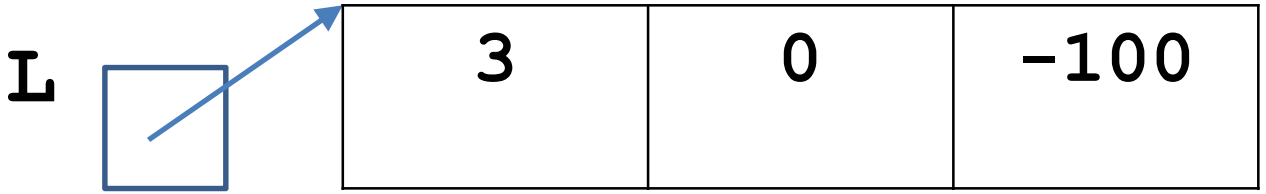
Other ways to mutate a list

```
L = [3,0]
```



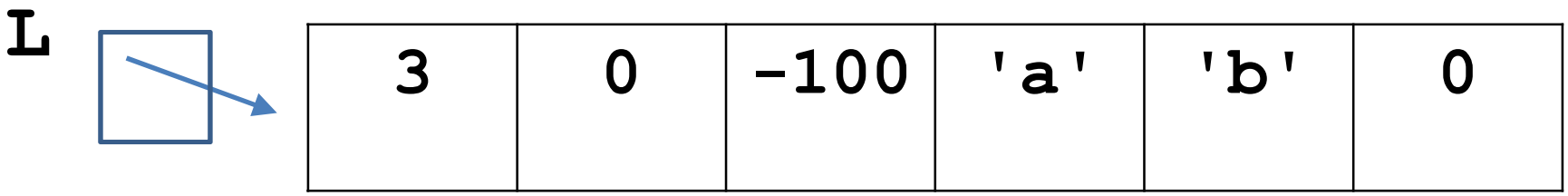

```
L.append(-100)
```

WARNING: Do NOT write
`L = L.append(-100)`
since **`append`** returns **`None`**.

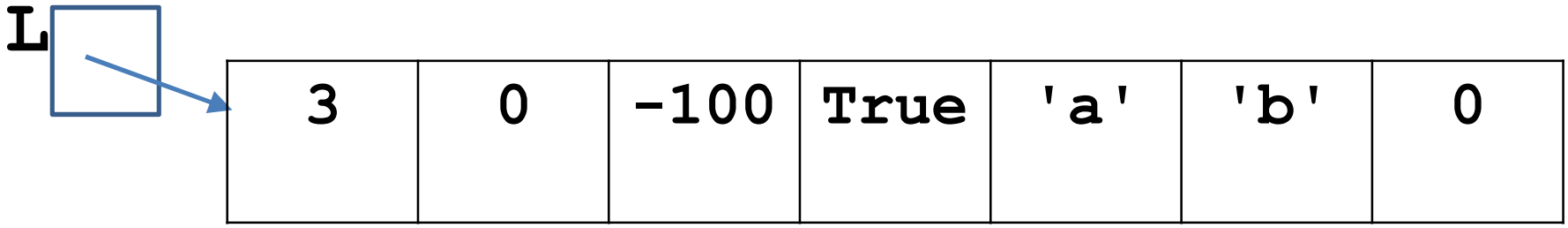


More ways to mutate a list

```
L.extend(['a','b',0])
```

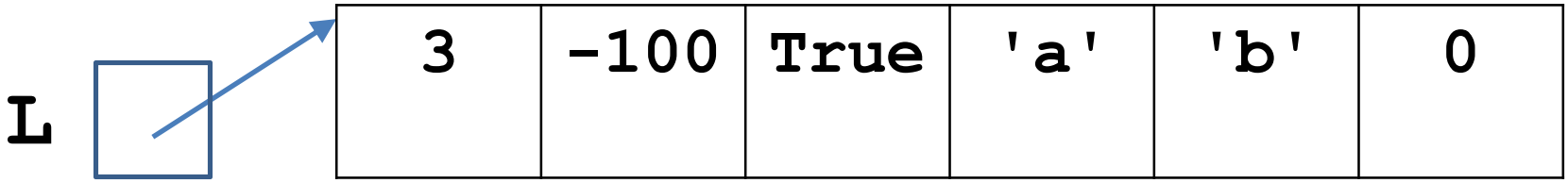



```
L.insert(3, True)
```

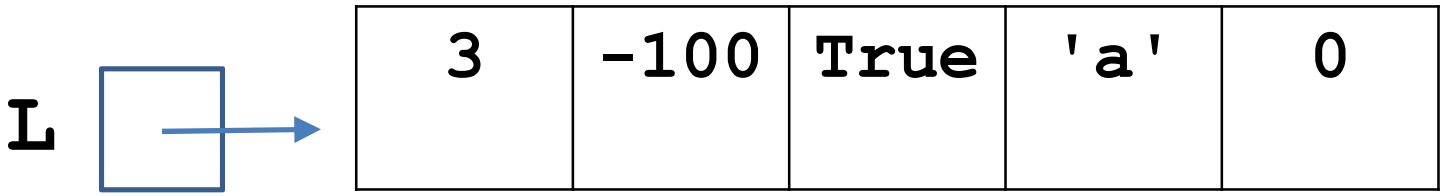


Even more ways to mutate a list

```
L.remove(0)
```

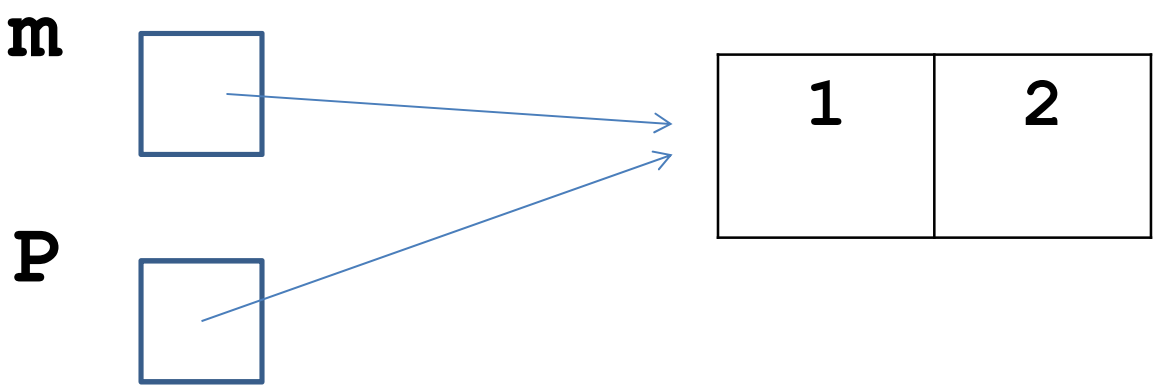



```
L.pop(4) ⇨ 'b'
```



Sharing list values

```
m = [1,2]
p = m
```

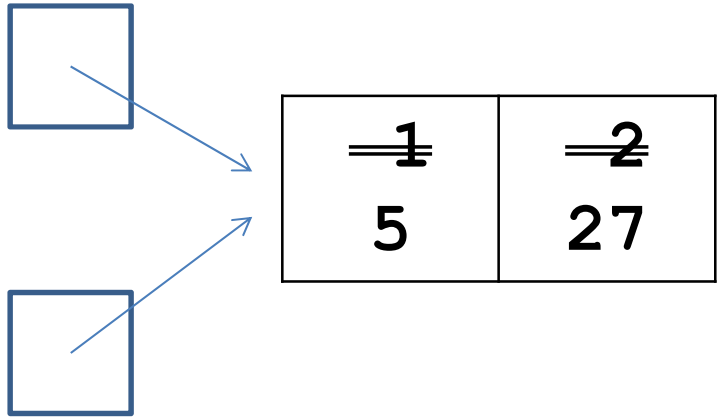


When two variables point to a common value in memory, they are called ***aliases***.

Consequences of aliasing

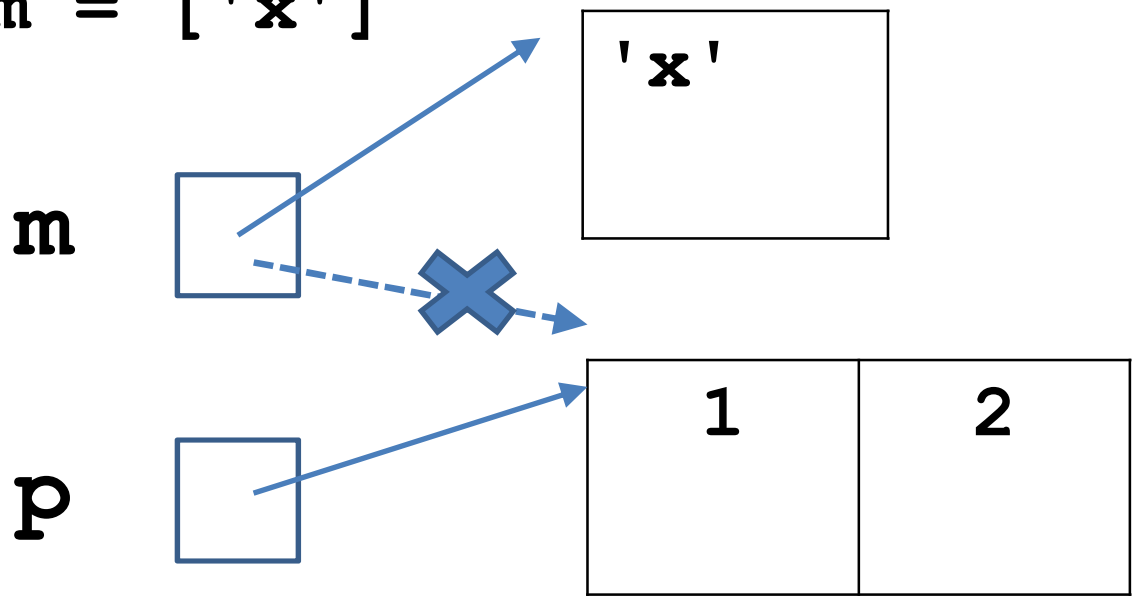
If two variables are aliases of a common list, you can mutate the existing list using either variable name

```
m = [1,2]
p = m
m[0] = 5
p[1] = 27
```



Can we break aliases?

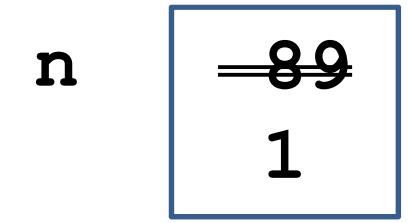
```
m = [1,2]
p = m
m = ['x']
```

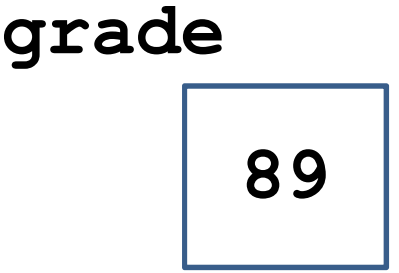


Yes! Just assign a new value to one of the variables. Only that variable is changed.

Recall: Functions and Immutable Parameters

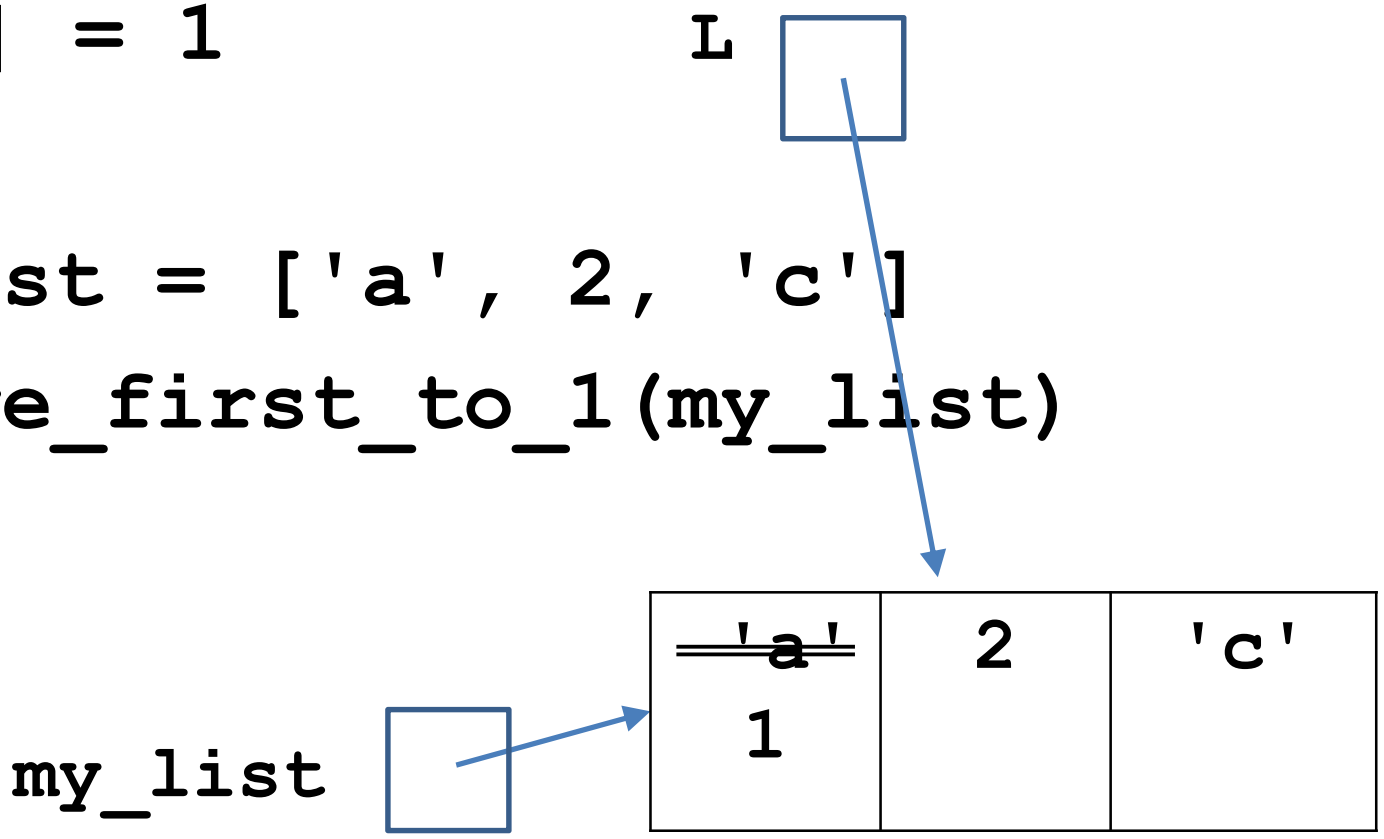
```
def change_to_1(n):
  n = 1

grade = 89
change_to_1(grade)
```





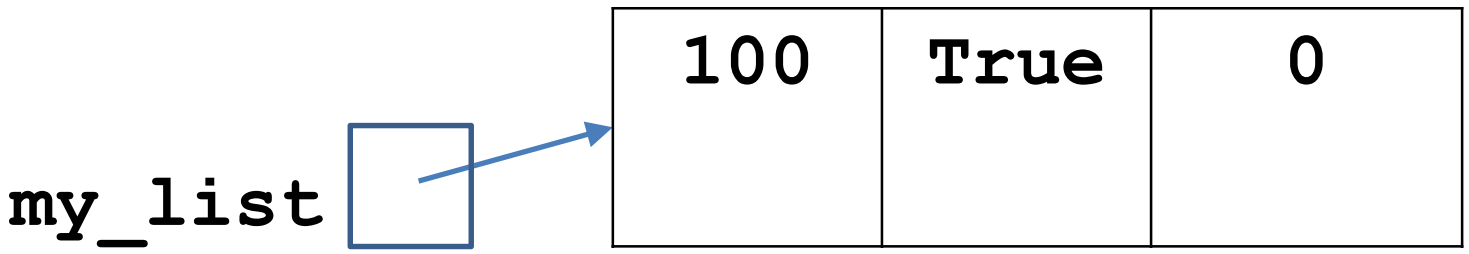
Functions and List Parameters

```
def change_first_to_1(L):
  L[0] = 1

my_list = ['a', 2, 'c']
change_first_to_1(my_list)
```



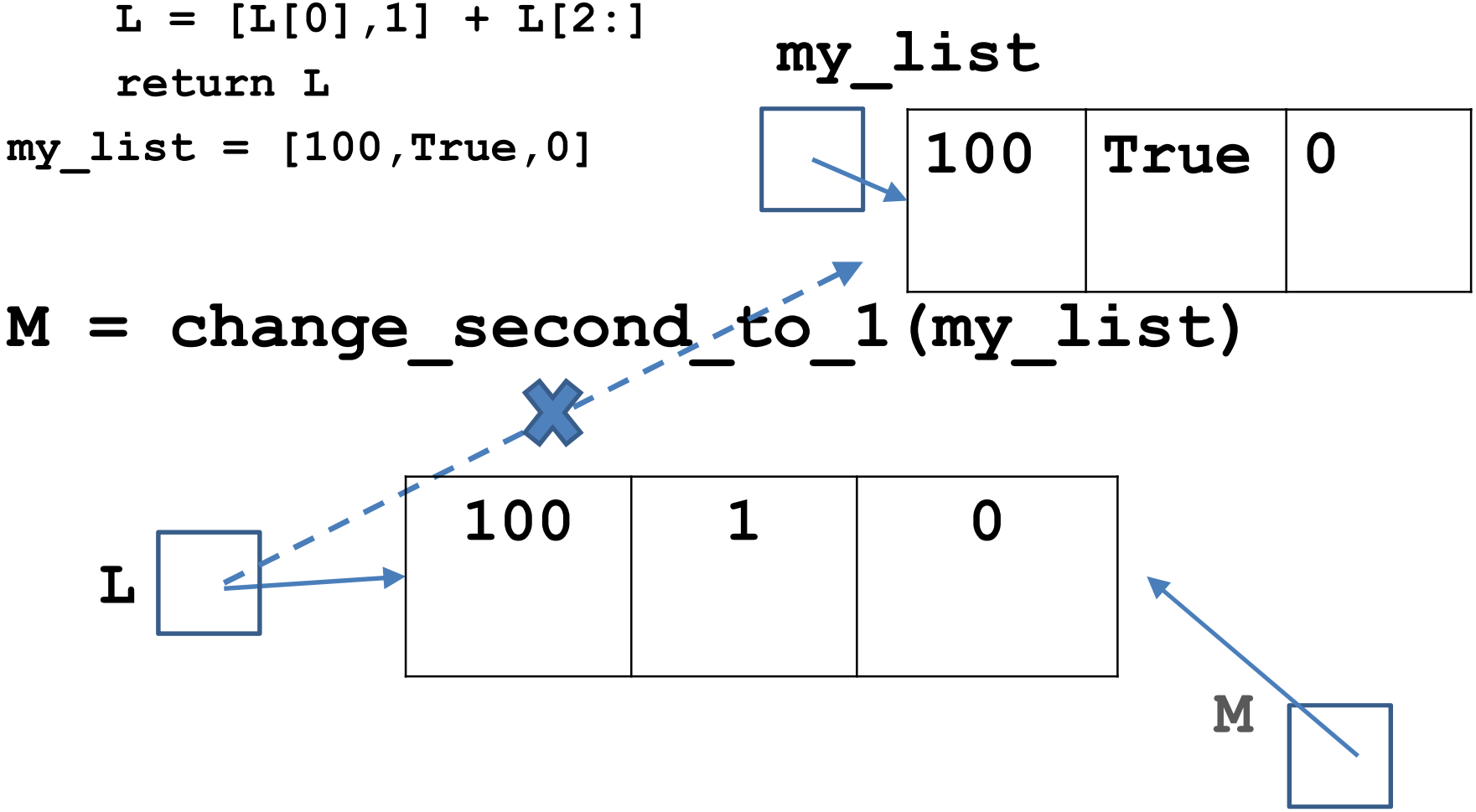
What is different here?

```
def change_second_to_1(L):
    L = [L[0],1] + L[2:]
    return L

my_list = [100,True,0]
```



What is different here?

```
def change_second_to_1(L):
    L = [L[0],1] + L[2:]
    return L
my_list = [100,True,0]
```

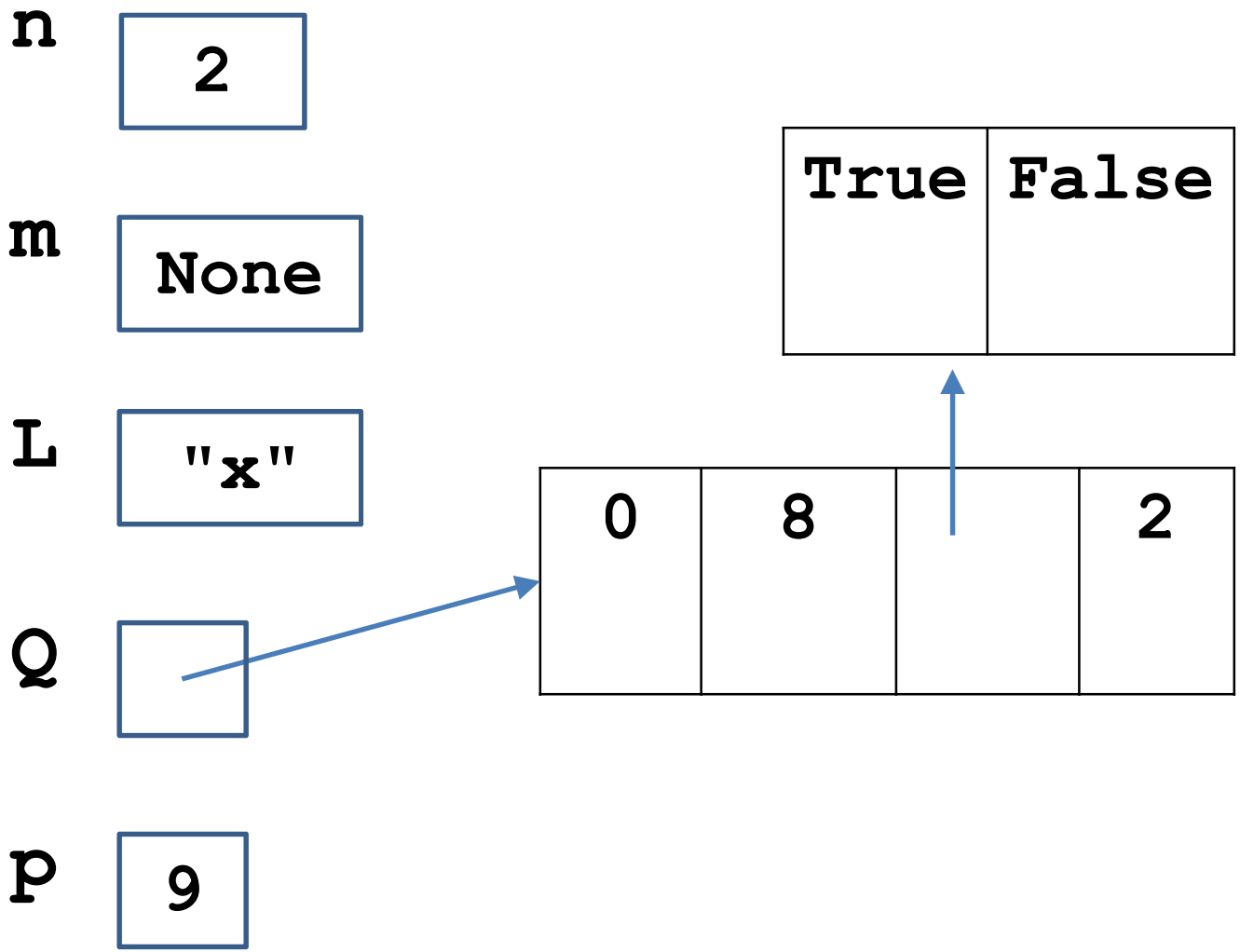


Exercise: Memory management

```
def dec(t):
    t = t - 1
    return t
```

```
n = 1
m = n
n = 2*m
L = [3,6,9]
Q = L
p = L[2]
Q[0] = dec(m)
L[1] = dec(Q[2])
L[2] = [True,
False]
m = L.append(n)
L = "x"
```

Completed exercise



When writing a function with lists

- Important to determine if a function is supposed to
  - Use the values in an existing list,
  - Mutate an existing list, or
  - Create and return a new list
- Review ThinkP 10.12

Design recipe additions

If a function mutates the contents of a parameter, this must be included in the Purpose and Effects statements.

Possible effects of a function:

- Printing to screen
- Reading from keyboard
- Mutation of parameter

Testing Mutation

For each test:

1. Set values of variables for testing
2. Call the appropriate **`check`** function to compare actual returned value to expected returned value (which might be **`None`**)
3. Call the appropriate **`check`** function on each testing variable that has been mutated, comparing the actual value to the expected value after mutation.

Example: Mutation

```
import check
import math

def multiply_first(L, factor):
    L[0] = L[0] * factor

## Test 1: factor = 0
L = [10,-2,3]
check.expect("T1", multiply_first(L,0), None)
check.expect("T1{L}", L, [0,-2,3])

## Test 2: factor not an integer (pi)
L = [10,0,-3.25]
check.expect("T2", multiply_first(L,math.pi), None)
check.within("T2(L)", L, [31.415926,0,-3.25],
0.00001)
```

Example: **`multiply_by`**

Use recursion to complete the Python function **`multiply_by`** that consumes a list of integers (**`vals`**) and another integer (**`multiplier`**) and mutates **`vals`** by multiplying each value in it by **`multiplier`**. The function returns **`None`**.

Example: `multiply_by`

```
def multiply_by(vals, multiplier):
    '''multiplies each value in vals by
       multiplier
       Effects: mutates vals

       multiply_by:(listof Int) Int-> None

       Example: for L = [1,2,3],
          multiply_by(L, 10) => None, and
          changes contents of L to
          [10,20,30]'''
```

Lists can be nested

Consider the list

```
L = [[1,2], [], [7,8,9,10]]
```

What are the values of:

```
len(L)
L[0]
len(L[1])
L[2][3]
```

How do we negate the entry 10?

Working with nested lists

```
def sum_firsts(lol):
    '''returns the sum of all the first values
         in the nonempty lists in lol
       sum_firsts: (listof (listof Int)) => Int
       Example: sum_firsts([[1,2],[],[7,8,9]])
                => 8
    '''
    if lol == []: return 0
    elif lol[0] == []:
        return sum_firsts(lol[1:])
    else:
        return lol[0][0] + sum_firsts(lol[1:])
```

Other Relevant List Information

- Retrieving the first element in a list is as fast as retrieving the last element
- In many other languages:
  - Lists are of a fixed size once created
  - Lists can only contain one type of value
  - Processing these lists (often called arrays) tends to be faster than processing Python lists
- Python has an **`array`** module (not used in CS116)

Functional Abstraction in Python

- Abstract list functions in Python 3 return iterators
- Use the functions **`map`** and **`filter`** as you did in Racket, but cast the value returned to a list.
  - **`list(map(fn, lst))`**
  - **`list(filter(fn, lst))`**

Functional Abstraction in Python: **map**

```
list(map(function, lst))
   returns a new list, applying
   function to each element in list
Requires: type consumed by function must
   match type in lst
-------------------------------------------------------------------
def pull_to_passing(mark):
    if mark < 50 and mark > 46:
        return 50
    else:
        return mark
list(map(pull_to_passing,
                 [34, 89, 46, 49, 52]))
⇨ [34, 89, 46, 50, 52]
```

Functional Abstraction in Python: `filter`

```
list(filter(function, lst))
  returns a new list matching the
  elements in list for which function
  returns True
Requires: type consumed by function must
  match type in lst
-------------------------------------------------------
def big_enough(mark):
    return mark>50
list(filter(big_enough,
                    [34, 89, 46, 49, 52]))
⇨ [89, 52]
```

Using `map` and `filter`

- Both consume a function and a list
- The type consumed by the parameter function must match the type of values in the parameter list
- Sometimes, we define a function and it is only used in a call to `map` or `filter`
- We can avoid this by using a `lambda` expression

`lambda`

- Like Racket, Python allows for anonymous functions using **`lambda`**
- Syntax:

```
lambda x: body
lambda x,y: body
```

- Note that **`body`** should be an expression, not a statement

Examples

```
def starters(words, start):
    return list(filter(lambda s:
                       s[:1]==start, words))

def expts_of(base, exponents):
    return list(map(lambda k: base**k,
                    exponents))
def cap_A(words):
    return list(map(lambda s: 'A'+s[1:],
                        list(filter(lambda s:
                          s[:1]=='a', words))))
```

Using **`range`** with **`map`** and **`filter`**

- To create a list we may use **`range`**:
  - **`list(range(6)) => [0,1,2,3,4,5]`**
  - **`list(range(3,7)) => [3,4,5,6]`**
  - **`list(range(4,15,3)) => [4,7,10,13]`**
- This is extra powerful with **`map`** and **`filter`**:

```
def sqr(x): return x*x
list(map(sqr, range(6)))
⇨ [0, 1, 4, 9, 16, 25]
```

`map` and **`filter`** can operate on strings as well

```
def just_digits(word):
    return "".join(list(filter(
        lambda c:c.isdigit(),
        word)))
```

Note: Even when processing strings, **`map`** and **`filter`** return an **`Iterator`**, which should be converted to a list. If you want a string answer, you'll likely need to use **`join`**.

Review: when functions consume lists as parameters

Function definition: **`def f(L):`** …

Function call: **`f(A)`**

- If **`f`** makes an assignment directly to **`L`**, then **`A`** is not changed. (**`L =`** …)
- If **`f`** makes an assignment to an element of L, or mutates L using list methods, then the contents of list referenced by A is changed as well. (**`L[0] =`** …, **`L.extend(`**…**`)`**)

```
def fn_two(L,M,x):
'''fn_two:(listof Y)(listof Z) X-> None'''
    x = 10
    L = "Howdy"
    M[0] = 'abc'
    M.append(x)

# Call the function
A = []
B = [1,2,3]
z = 42.42
fn_two(A,B,z)
print(A, B, z)
```

Goals of Module 04

- We should now be able to write any of our Racket programs in Python, using
  - Lists and their methods
  - Lists used to implement structures
  - Mutation of lists
  - Functional abstraction and **`lambda`**