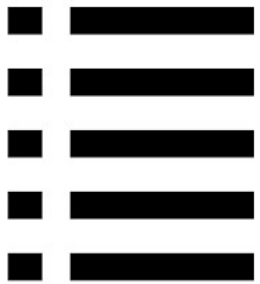
Lecture 05: Lists

Stephen Huang

March 15, 2023



Contents

1. Python Data Structures

- Data structures are structures that can hold some data together. In other words, they are used to store a collection of related data.
- There are four built-in data structures in Python
  - list,
  - tuple,
  - dictionary, and
  - set.
- We will spend more time on lists, arguably the most useful ones. Many, but not all, of the discussions on lists apply to the other three.

Overview

Types	Ordered	Indexed	Collection Changeable?	Item Changeable?	Duplicate
List	Yes	Yes	Add/Remove	Yes	Yes
Tuple	Yes	Yes	No	No	Yes
Set	No	No	Add/Remove	No	No
Dictionary	No	Yes	Yes	Value Yes Key No	No

List Data Structure

- A list is a sequence of elements (0 or more). A list can be empty.
- A list is also called an array.
- A list is a data structure that can be decomposed into multiple elements.
- In most other languages, list elements must be homogenous (of the same type).
- In Python, the elements can be heterogeneous (of different types).

List

- Some types of a list: string, int, float, etc.
- An element of a list can be a list. So, we can have a list of lists or a nested list.
- Since multiple elements may exist in a list, each element is uniquely identified by its position.
- Positions start with 0, 1, 2, …
- To access the i-th element of a list x, use x[i].

Visualization

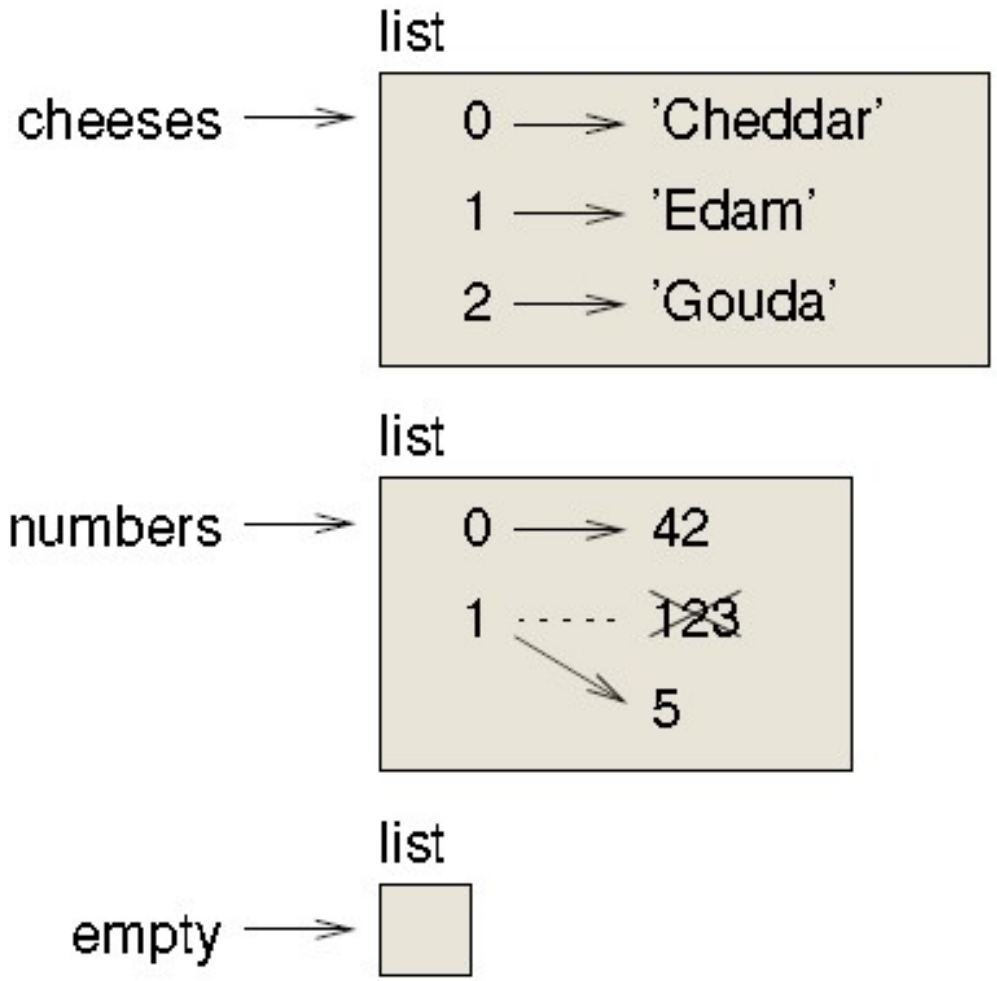


Figure 10.1: State diagram.

2. Defining Lists

- There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]).
  - [10, 20, 30, 40]
  - [“apple”, “mango”, “banana”]
  - [“apple”, 20, 30.5]
  - []
- An element can be of any type, including a list itself.
  - [20, 30, [1, 2]]

Examples

```
>>>
>>> list1 = [apple]
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    list1 = [apple]
NameError: name 'apple' is not defined
>>> list1 = ['a', 'b']
>>> list2 = [list1]
>>> list2
[['a', 'b']]
>>>
```

Definition

- In general, a list element can be an expression. Thus, it should be evaluated first. The result is then used as a list element.

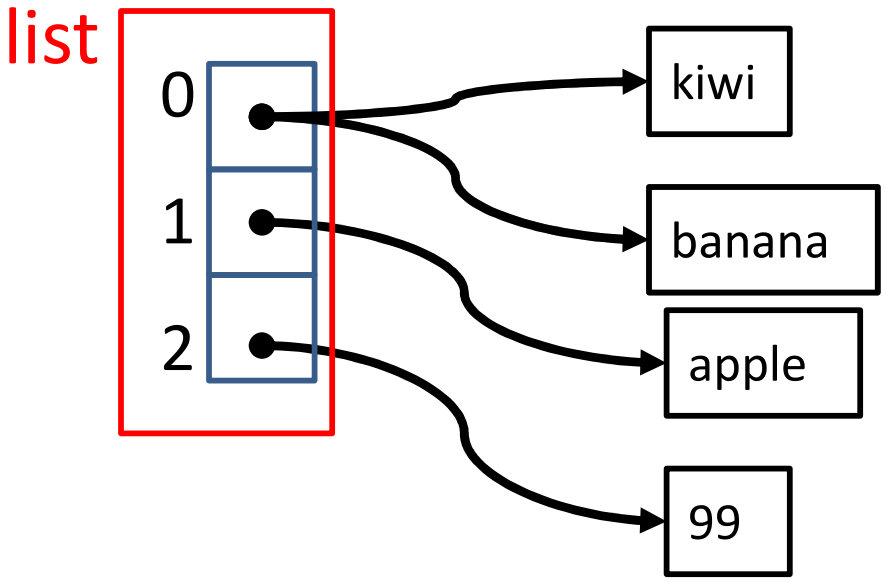
```
>>> x = 100
>>> str = "UH"
>>> list1 = [9**2, x/3, str*2]
>>> list1
[81, 33.333333333333336, 'UHUH']
>>>
```

```
>>> a = [9]
>>> b = [a, 99]
>>> c = [b, a, 999]
>>> print(a,b,c)
[9] [[9], 99] [[[9], 99], [9], 999]
>>>
```

Lists are Mutable

```
>>>
>>> numbers = [1, 3, 5, 7, 9]
>>> numbers
[1, 3, 5, 7, 9]
>>> numbers[3]
7
>>> numbers[0]
1
>>> numbers[2] = 99
>>> numbers
[1, 3, 99, 7, 9]
>>>
...
```

Lists are mutable



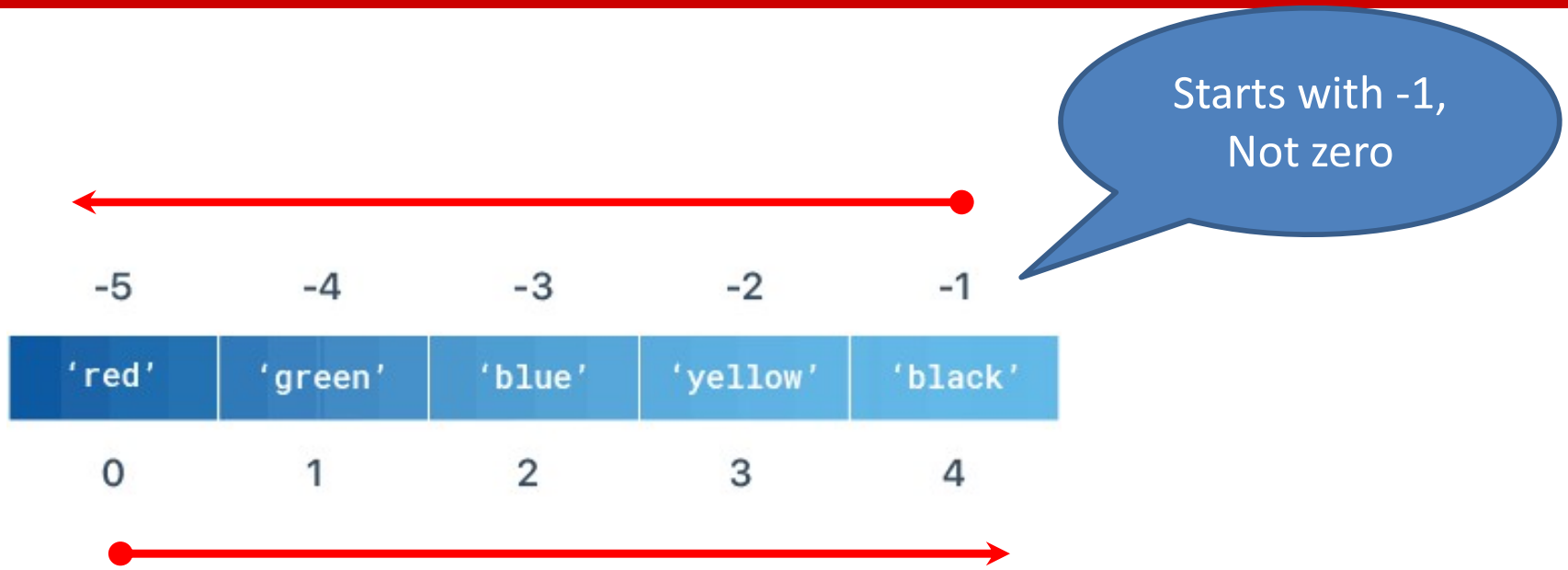
Empty list

List index

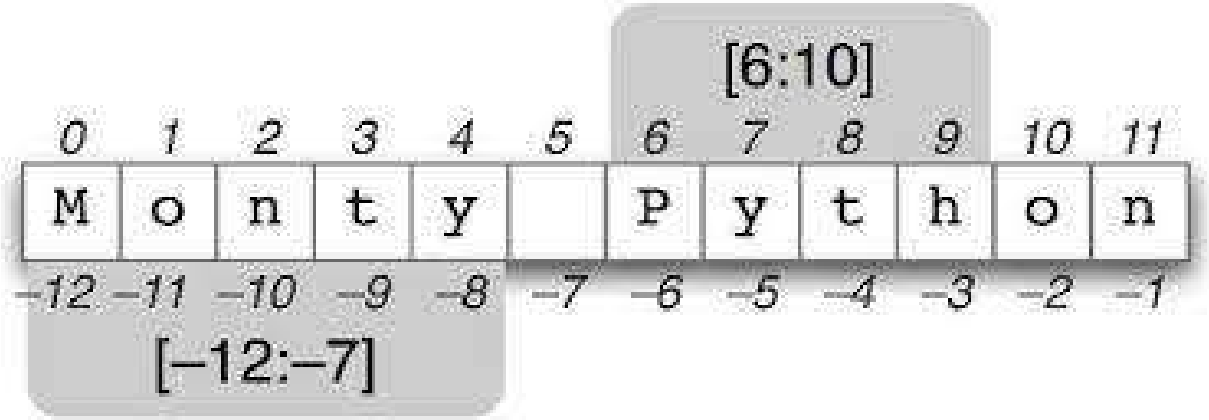
- List indices work the same way as string indices:
  - Any integer expression can be used as an index.
  - If you try to read or write an element that does not exist, you get an `IndexError`.
  - If an index has a negative value, it counts backward from the end of the list.

Negative index

- List



- String



Membership Operator

- The `in`-operator works on lists.

```
>>> fruits = ['banana', 'apple', 'mango', 'pear']
>>> 'apple' in fruits
True
>>> 'Apple' in fruits
False
>>> 'peach' in fruits
False
```

3. List Enumeration

- Python's built-in enumerate function allows us to loop over a list and retrieve both the index and the value of each item in the list.
- The enumerate function gives us an iterable where each element is a tuple containing the item's index and the original item value.
- Syntax: `enumerate(<iterable>, start)`
- The `start` is optional and defaults to 0.

Example

```
fruits = ['apple', 'banana', 'mango',
          'pear', 'watermelon']

i = 0
while i < len(fruits):
    print(f'{i}: {fruits[i]}')
    i += 1
```

```
0: apple
1: banana
2: mango
3: pear
4: watermelon
```

Example

```
fruits = ['apple', 'banana', 'mango',
          'pear', 'watermelon']

for i in range(len(fruits)):
    print(f'{i}: {fruits[i]}')
```

0: apple
1: banana
2: mango
3: pear
4: watermelon

Example

```
fruits = ['apple', 'banana', 'mango',
          'pear', 'watermelon']

for item in enumerate(fruits):
    print(item)
```

```
(0, 'apple')
(1, 'banana')
(2, 'mango')
(3, 'pear')
(4, 'watermelon')
```

Example

```
fruits = ['apple', 'banana', 'mango',
          'pear', 'watermelon']

for index, fruit in enumerate(fruits):
    print(f'{index}: {fruit}')
```

0: apple
1: banana
2: mango
3: pear
4: watermelon

Example

```
fruits = ['apple', 'banana', 'mango',
          'pear', 'watermelon']

for index, fruit in enumerate(fruits, 1):
    print(f'{index}: {fruit}')
```

1: apple
2: banana
3: mango
4: pear
5: watermelon

4. Traversing a list

- It is a common practice to "visit" every list element sequentially.
- "Traversal."

```
>>> fruits = ['banana', 'apple', 'mango', 'pear']
>>> for f in fruits:
        print(f)

banana
apple
mango
pear
>>>
```

List

```
>>> prime = [2, 3, 5, 7, 11, 13, 17, 19]
>>> for p in prime:
        print(p)

2
3
5
7
11
13
17
19
>>> for i in range(len(prime)):
        print(i,": ", prime[i], sep='')

0: 2
1: 3
2: 5
3: 7
4: 11
5: 13
6: 17
7: 19
```

List Traversal

```
prime_list = [2, 3, 5, 7, 11, 13, 17, 19]

for p in prime_list:
    print(p)

for i in range(len(prime_list)):
    print(i, ':', prime_list[i])

for i, p in enumerate(prime_list, 1):
    print(i, ':', p)
```

Print

```
prime_list = [2, 3, 5, 7, 11, 13, 17, 19]

for p in prime_list:
    print(p)
```

```
2
3
5
7
11
13
17
19
```

List Index

```
# use the index to get the value
for i in range(len(prime_list)):
    print(i, ':', prime_list[i])

# use enumeration to get the index
for i, value in enumerate(prime_list):
    print(i, ':', value)
```

```
0 : 2
1 : 3
2 : 5
3 : 7
4 : 11
5 : 13
6 : 17
7 : 19
```

List Operators

```
a = [1,2,3]
b = [4,5,6]
c = a + b    # concatenation
print(b*3)   # repetition

a = []
for i in range(10):
    a = a+[i]    # concatenation

if x in b:
   print x
```

Example: Max

```
def list_max(alist):
    max = alist[0]
    for elem in alist:
        if elem>max:
            max = elem

    return max

a = [2, 3, 25, 4, 9, 8, 7, 16, 25]
print(list_max(a))
```

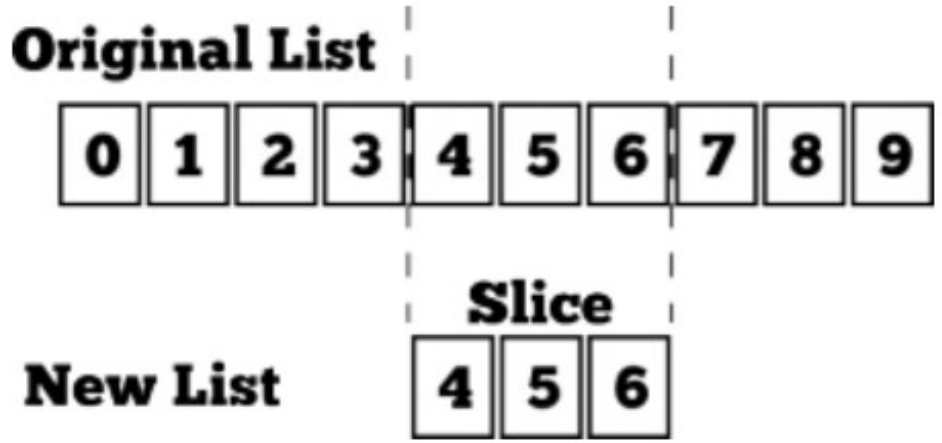
Where is the max?

Example: Max

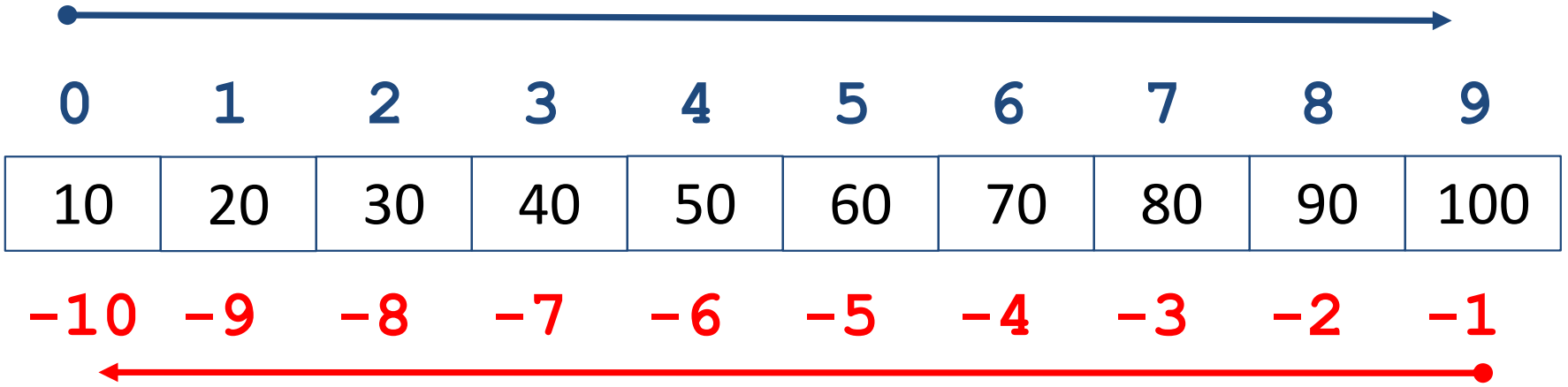
```
def list_max(a):
    max = a[0]
    index = 0
    for i, elem in enumerate(a):
        if elem>max:
            max = elem
            index = i
    return index

a = [2, 3, 25, 4, 9, 8, 7, 16, 25]
idx = list_max(a)
print(idx, ":  ", a[idx])
# There is a max() for list
```

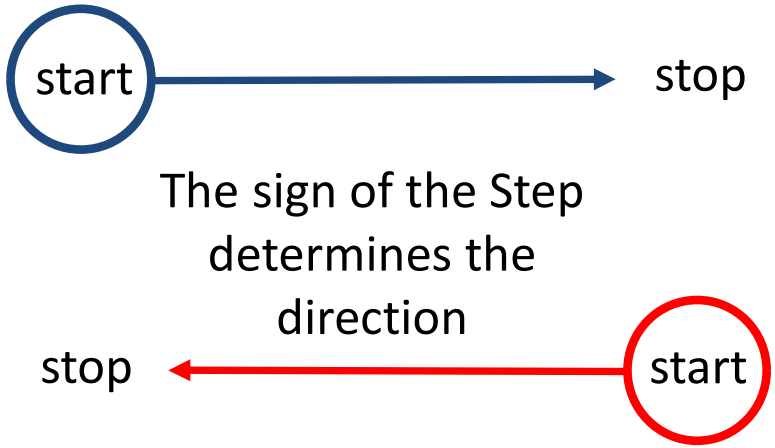
5. List Slices

- Slicing is the ability to create a list from another list by cutting pieces of that other list.
- The new list is a different copy.
- The original list is unchanged.
- Three parameters: start, stop, and step.
- `list[start:stop:step]`, any one can be omitted.



Negative Indices





Example

```
L = list(range(10))
print(L)
low = 1
high = 8
print(L[low:high:2])
print(L[high:low:-1])
print(L[high:low:-2])


a = [1,2,3]
a[1:3] = [4, 5, 6]
print(a)
```

Slice Examples

```
a=[1,2,3]
a[1:3] = [4, 5, 6]
print(a)
```

[1, 4, 5, 6]

```
L = list(range(10))
b = L[:7]
print(b)
```

[0, 1, 2, 3, 4, 5, 6]

```
b = L[3:]
print(b)
```

[3, 4, 5, 6, 7, 8, 9]

```
b = L[3:7]
print(b)
```

[3, 4, 5, 6]

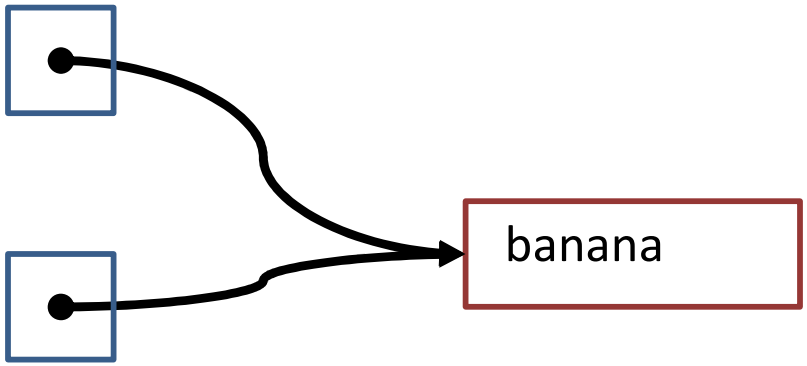
```
b = L[3:-2]
print(b)
```

[3, 4, 5, 6, 7]

"Pointers"

a = "banana"

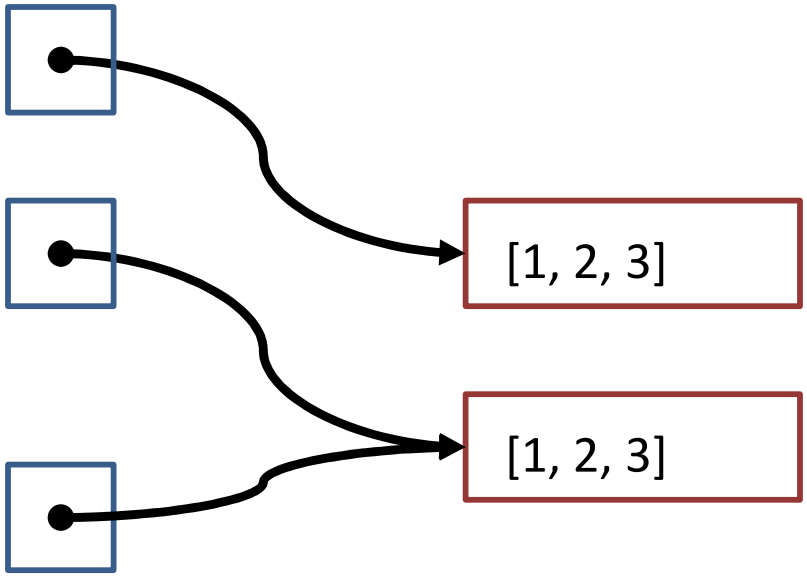
b = "banana"



A = [1, 2, 3]

B = [1, 2, 3]

C = B

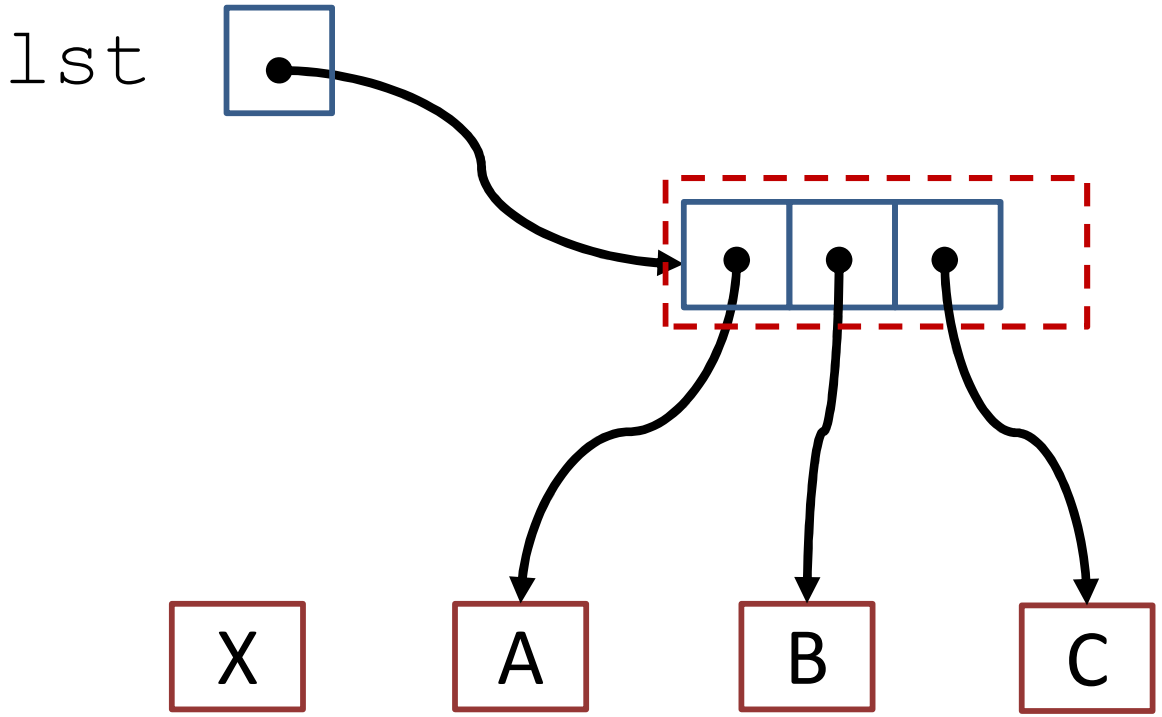


Alias

- In the example of strings, Python only created one string object, both `a` and `b`. But when you create two lists (`A` and `B`), you get two objects.
- In this case, we would say that the two lists are equivalent because they have the same elements but are not identical.
  - Identical => equivalent.
- If `a` refers to an object and you assign `b = a`, then both variables refer to the same object. The variable `b` is an alias of `a`.

Visualization of a list

```
lst = ['a', 'b', 'c']
```



6. List Methods

- Python provides many methods that operate on lists.
- Most list methods are void; they modify the list and return **`None`**.
- Remember, lists are mutable.

Methods

- append(...)

  L.append(object) -> None -- append object to end
- clear(...)

  L.clear() -> None -- remove all items from L
- copy(...)

  L.copy() -> list -- a shallow copy of L
- count(value)

  L.count(value) -> integer -- return number of occurrences of value
- extend(...)

  L.extend(iterable) -> None -- extend the list by appending elements from the iterable

Methods

- index(...)

  L.index(value, [start, [stop]]) -> integer -- return the first index of value.

  Raises ValueError if the value is not present.

- insert(...)

  L.insert(index, object) -- insert object before index

- pop(...)

  L.pop([index]) -> item -- remove and return the item at index (default last).

  Raises IndexError if the list is empty or the index is out of range.

- remove(...)

  L.remove(value) -> None -- remove the first occurrence of value.

  Raises ValueError if the value is not present.

Methods

- reverse(...)

  L.reverse() -- reverse *IN PLACE*

- sort(...)

  L.sort(key=None, reverse=False) -> None -- stable sort *IN PLACE*

Methods

- You can remove an element by calling remove or delete.
  - `L.remove('b')` remove 'b' from the list. If there is more than one copy of 'b', only one is removed. By-value
  - `del L[i]` deletes the element at the i-th position of the list. By-index
  - You can also do `x = L.pop(i),` which pops off the i-th element and put that in `x`.

Positions are relative

- Lists are mutable in Python.
- Positions are relative.
- When we make changes (pop, remove, insert) to a list, the position of an element may be changed.
  - Suppose you ranked #3 in the class, and James dropped out of the course. What is your rank?
  - Your rank changes even though you did not do anything.

John
James
You
….

Example

```
list = list(range(8))
list.insert(3,99)
print(list)
```

```
[0, 1, 2, 99, 3, 4, 5, 6, 7]
```

```
list = list(range(8))
for i in range(len(list)):
    if list[i]==3:
        list.pop(i)
    else:
        print(list[i],", ",sep='',end='')
```

```
0, 1, 2, 5, 6, 7, Crash!!!
```

Example

```
list = list(range(8))
for item in list:
    if item==3:
        list.remove(3)
    else:
        print(item)
```

0, 1, 2, 5, 6, 7 why?

Improved Version

```
list = list(range(8))
i = 0
while i<len(list):
    if list[i]==3:
        list.pop(i)
    else:
        print(list[i],", ",sep='',end=''
    i+=1
```

0, 1, 2, 5, 6, 7, Crash!!!

Final Version

```
list = list(range(8))
i = 0
while i<len(list):
    if list[i]==3:
        list.pop(i)
    else:
        print(list[i],", ",sep='',end=''
        i+=1
```

0, 1, 2, 4, 5, 6, 7, ☺

7. Multi-Dimensional Lists

- A list of lists is a multi-dimensional list or multi-dimensional array.
- One can access a multidimensional array using multiple indices like `a[i][j]`.
  - Not `a[i,j]`.
- The order of the index is essential. The first refers to the index of the outer list, and the second relates to the inner list.
- Just like a one-dimensional array, a list must be created before you can use it.

Creating a 1D list

```
def create(n):
    list = []
    for i in range(n):
        list.append(i)
    return list
```

Creating a 2D list

```
def create(m,n):
    list = []
    for i in range(m):
        sublist=[]
        for j in range(n):
            sublist.append(100*i+j)
        list.append(sublist)
    return list
```

Traversing a 1D list

```
def printlist(list, x):
    for elem in list:
        elem=elem*x
        print('{elem:4d}', end=' ')
    print()

list = create(5)
printlist(list, 10)
printlist(list, 2)
```

Traversing a 2D list

```
def printlist(list):
    for sublist in list:
        for elem in sublist:
            print(f'{elem:5d}', end=' ')
        print()
    print()
```

Traversing a 2D list

```
def printlist(list):
    for i in range(len(list)):
        for j in range(len(list[i])):
            print('{list[i][j]:5d}', end=' ')
        print()
    print()
```

8. List Comprehension

- In Math, we sometimes use this to define a set:

$$\{x^2 | 1 \leq x \leq 6, x \text{ is odd}\}$$

- It is straightforward to understand what the set is. Of course, we are dealing with lists here. So, imagine we have a list of all integers.
- It would be nice if we could do it on a list.
  - Select only the odd numbers between 1 and 6
  - Transform the numbers into their squares
  - Make them into a list.

What is it?

- List comprehension allows us to make a new list from a list.
- List comprehension provides a syntax for transforming one list into another list.
- Elements can be conditionally included (only odd numbers) in the new list, and each element can be transformed (square) as needed.
- You don't have to use a list comprehension. A (for-) loop can do the same job.
- However, list comprehension is easier to understand the code's intention.

Solutions

```
numbers = [1, 2, 3, 4, 5, 6]
half_evens = []
for n in numbers:
    if n%2 == 0:
        half_evens.append(n/2)

half_evens = [
    n/2 for n in numbers if n%2==0
]
```

I don't see why is this better.
It's difficult to understand.

Solutions

```
numbers = [1, 2, 3, 4, 5, 6]

half_evens = [
    n/2
    for n in numbers
    if n%2 == 0
]
```

Now I see it.

$$\{\frac{n}{2} | 1 \leq n \leq 6, n \text{ is even}\}$$

Template

```
new_list = []
for item in ori_list:
    if condition(item):
        new_list.append(expression(item))
```

Constructive:
How to generate the list

```
new_list = [
    expression(item)
    for item in ori_list
    if condition(item)
]
```

Definitional:
What the list is

You don't have to use it if you don't like it.

Other Comprehensions

- This applies to other structures too
  - Set comprehension
  - Dictionary comprehension