Lists

•A list is an ordered set of values, where each value is identified by an index.

The values that make up a list are called its elements.

Lists are similar to strings, which are ordered sets of characters, except that the elements of a list can have any type. Lists and strings and other things that behave like ordered sets are called sequences.

•List values

•There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

[10, 20, 30, 40]

["spam", "bungee", "swallow"]

The first example is a list of four integers.

The second is a list of three strings.

•The following list contains a string, a oat, an integer, and (mirabile dictu) another list:

["hello", 2.0, 5, [10, 20]]

A list within another list is said to be nested.

Lists that contain consecutive integers are common, so Python provides a simple way to create them:

>>> range(1,5)

•The range function takes two arguments and returns a list that contains all the integers from the first to the second, including the first but not including the second!

There are two other forms of range. With a single argument, it creates a list that starts at 0:

>>> range(10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

•**If** there is a third argument, it species the space between successive values, which is called the step size. This example counts from 1 to 10 by steps of 2:

>>> range(1, 10, 2)

[1, 3, 5, 7, 9]

•**Finally,** there is a special list that contains no elements. It is called the empty list, and it is denoted [].

With all these ways to create lists, it would be disappointing if we couldn't assign list values to variables or pass lists as arguments to functions. We can. **For example:**

vocabulary = ["ameliorate", "castigate", "defenestrate"]

numbers = [17, 123]

empty = []

print vocabulary, numbers, empty

['ameliorate', 'castigate', 'defenestrate'] [17, 123] []

Accessing elements

•The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string the bracket operator ([]). The expression inside the brackets species the index. Remember that the indices start at 0:

print numbers[0]

numbers[1] = 5

•The bracket operator can appear anywhere in an expression. When it appears on the left side of an assignment, it changes one of the elements in the list, so the one-eth element of numbers, which used to be 123, is now 5.

•Any integer expression can be used as an index:

>>> numbers[3-2]

5

>>> numbers[1.0]

TypeError: sequence index must be integer

If you try to read or write an element that does not exist, you get a runtime error:

```
>>> numbers[2] = 5
```

IndexError: list assignment index out of range.

If an index has a negative value, it counts backward from the end of the list:

```
>>> numbers[-1]
5
>>> numbers[-2]
17
>>> numbers[-3]
IndexError: list index out of range
```

numbers[-1] is the last element of the list, numbers[-2] is the second to last, and numbers[-3] doesn't exist.

It is common to use a loop variable as a list index. **For example:**

```
horsemen = ["war", "famine", "pestilence", "death"]
i = 0
while i < 4:
print horsemen[i]
i = i + 1
```

This while loop counts from 0 to 4. When the loop variable **i** is 4, the condition fails and the loop terminates. So the body of the loop is only executed when **i** is 0, 1, 2, and 3.

Each time through the loop, the variable **i** is used as an index into the list, printing the **i-eth** element. This pattern of computation is called a list traversal.

List length

The function **len** returns the length of a list. It is a good idea to use this value as the upper bound of a loop instead of a constant. That way, if the size of the list changes, you won't have to go through the program changing all the loops; they will work correctly for any size list. **For example:**

```
horsemen = ["war", "famine", "pestilence", "death"]

i = 0

while i < len(horsemen):

print horsemen[i]

i = i + 1
```

The last time the body of the loop is executed, **i** is **len(horsemen) - 1**, which is the index of the last element. When **i** is equal to **len(horsemen),** the condition fails and the body is not executed, which is a good thing, because **len(horsemen)** is not a legal index.

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

['spam!', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]

List membership

in is a boolean operator that tests membership in a sequence (will use in next chapter). **For example:**

```
>>> horsemen = ['war', 'famine', 'pestilence', 'death']
>>> 'pestilence' in horsemen
True
>>> 'debauchery' in horsemen
False
```

Since **"pestilence"** is a member of the horsemen list, the in operator returns **true.** Since **"debauchery"** is not in the list, in returns **false**.

We can use the **not** in combination with **in** to test whether an element is not a member of a list:

```
>>> 'debauchery' not in horsemen
True
```

Lists and for loops

The generalized syntax of a for loop is:

for VARIABLE in LIST:

BODY

This statement is equivalent to:

```
i = 0
while i < len(LIST):
VARIABLE = LIST[i]
BODY
i = i + 1
```

The for loop is more concise because we can eliminate the loop variable, **i.** Here is the previous loop written with a for loop. **For example:**

```
for horseman in horsemen:
    print horseman
```

It almost reads like English: \For (every) horseman in (the list of) horsemen, print (the name of the) horseman."

Any list expression can be used in a for loop. **For example:**

```
for number in range(20):
    if number % 2 == 0:
        print number
for fruit in ["banana", "apple", "quince"]:
    print "I like to eat " + fruit + "s!"
```

The **first example** prints all the even numbers between zero and nineteen. The **second example** expresses enthusiasm for various fruits.

List operations

The **+** operator concatenates lists:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

Similarly, the * operator repeats a list a given number of times. **For example:**

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The **first example** repeats [0] four times. The **second example** repeats the list [1, 2, 3] three times.

•List slices

The slice operations also work on lists. **For example:**

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> list[1:3]
['b', 'c']
>>> list[:4]
['a', 'b', 'c', 'd']
>>> list[3:]
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end.

So if you omit both, the slice is really a copy of the whole list**. For example:**

```
>>> list[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

•Lists are mutable

Unlike strings, lists are mutable, which means we can change their elements.

Using the bracket operator on the left side of an assignment, we can update one of the elements. **For example:**

```
>>> fruit = ["banana", "apple", "quince"]
>>> fruit[0] = "pear"
>>> fruit[-1] = "orange"
>>> print fruit
['pear', 'apple', 'orange']
```

•With the slice operator we can update several elements at once. **For example:**

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> list[1:3] = ['x', 'y']
>>> print list
['a', 'x', 'y', 'd', 'e', 'f']
```

•We can also remove elements from a list by assigning the empty list to them. **For example:**

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> list[1:3] = []
>>> print list
['a', 'd', 'e', 'f']
```

And we can add elements to a list by squeezing them into an empty slice at the desired location. For example:

```
>>> list = ['a', 'd', 'f']

>>> list[1:1] = ['b', 'c']

>>> print list

['a', 'b', 'c', 'd', 'f']

>>> list[4:4] = ['e']

>>> print list

['a', 'b', 'c', 'd', 'e', 'f']
```

List deletion

Using slices to delete list elements can be awkward, and therefore error-prone. Python provides an alternative that is more readable. **del** removes an element from a list. **For example:**

```
>>> a = ['one', 'two', 'three']
>>> del a[1]
>>> a
['one', 'three']
```

As you might expect, **del** handles negative indices and causes a runtime error if the index is out of range. You can use a slice as an index for del. For example:

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del list[1:5]
>>> print list
['a', 'f']
```

As usual, slices select all the elements up to, but not including, the second index.

Objects and values

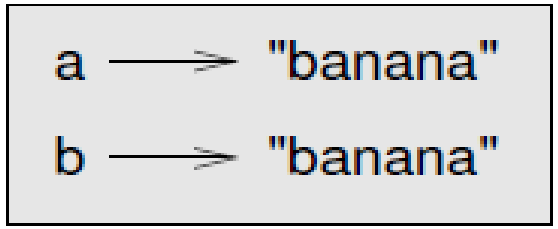
If we execute these assignment statements,

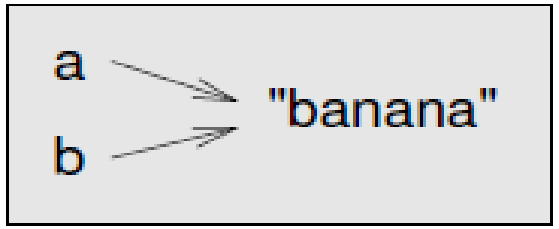
a = "banana"

b = "banana"

we know that **a** and **b** will refer to a string with the letters **"banana".** But we can't tell whether they point to the same string.

There are two possib





In one case, **a** and **b** refer to two different things that have the same value. In the second case, they refer to the same thing. These “things" have names they are called **objects.** An object is something a variable can refer to.

Every object has a unique identifier, which we can obtain with the **id** function.

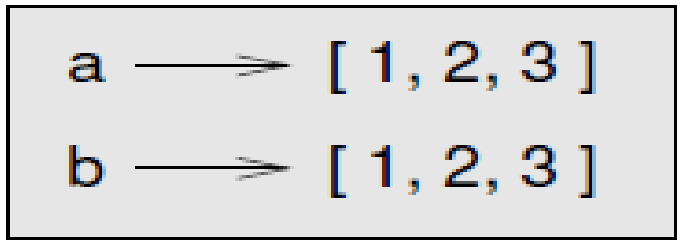
By printing the identifier of a and b, we can tell whether they refer to the same object. **For example:**

```
>>> id(a)
135044008
>>> id(b)
135044008
```

In fact, we get the same identifier twice, which means that Python only created one string, and both **a** and **b** refer to it.

Interestingly, lists behave differently. When we create two lists, we get two objects. **For example:**

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
135045528
>>> id(b)
```



a and **b** have the same value but do not refer to the same object.

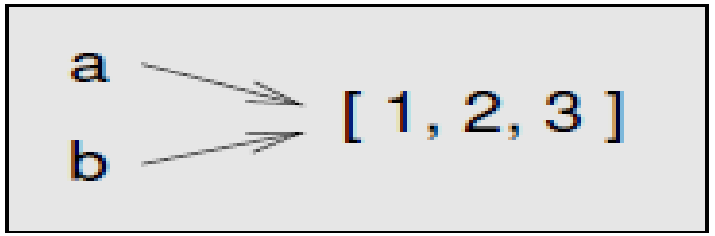
Aliasing

Since variables refer to objects, if we assign one variable to another, both variables refer to the same object. **For example:**

```
>>> a = [1, 2, 3]

>>> b=a
```

In this case, the state diagram looks like this:



Because the same list has two different names, a and b, we say that it is aliased.
Changes made with one alias affect the other. **For example:**

```
>>> b[0] = 5
>>> print a
[5, 2, 3]
```

Cloning lists

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called cloning, to avoid the ambiguity of the word "copy".

The easiest way to clone a list is to use the slice operator. For example:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> print b
[1, 2, 3]
```

Taking any slice of **a** creates a new list. In this case the slice happens to consist of the whole list.

Now we are free to make changes to **b** without worrying about **a**. **For example:**

```
>>> b[0] = 5
>>> print a
[1, 2, 3]
```

List parameters

Passing a list as an argument actually passes a reference to the list, not a copy of the list. For example, the function head takes a list as an argument and returns the first element. **For example:**

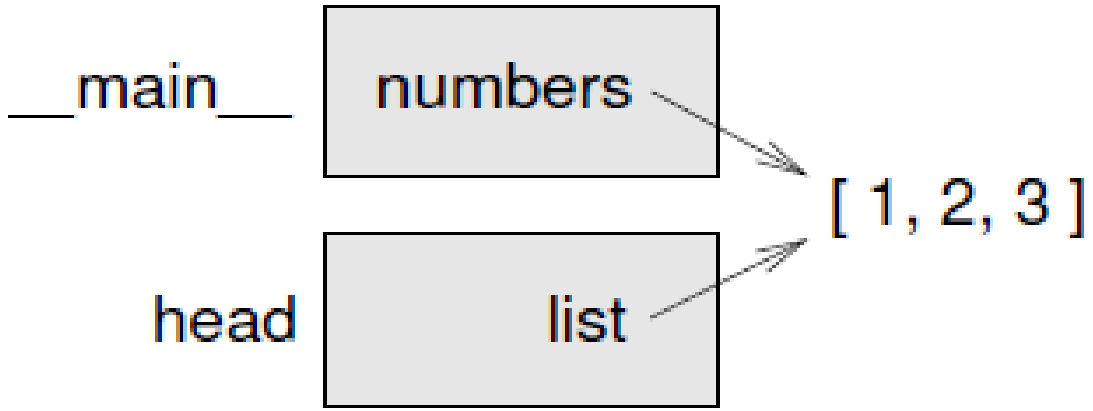
```
def  head(list):
  return list[0]
```

Here's how it is used:

```
>>> numbers = [1, 2, 3]
>>> head(numbers)
1
```

The parameter **list** and the variable **numbers** are aliases for the same object.

The state diagram looks like this:



Since the list object is shared by two frames, we drew it between them.
If a function modifies a list parameter, the caller sees the change. **For example:**
deleteHead removes the first element from a list:

```
def deleteHead(list):
del list[0]
```

Here's how **deleteHead** is used:

```
>>> numbers = [1, 2, 3]
>>> deleteHead(numbers)
>>> print numbers
[2, 3]
```

If a function returns a list, it returns a reference to the list. **For example,** tail returns a list that contains all but the first element of the given list:

```
def tail(list):
return list[1:]
```

Here's how tail is used:

```
>>> numbers = [1, 2, 3]
>>> rest = tail(numbers)
>>> print rest
[2, 3]
```

Because the return value was created with the **slice operator,** it is a new list. Creating rest, and any subsequent changes to rest, have no effect on numbers.

Nested lists

A nested list is a list that appears as an element in another list. In this example of list, the three-eth element is a nested list:

```
>>> list = ["hello", 2.0, 5, [10, 20]]
```

If we print list[3], we get [10, 20].

To extract an element from the nested list, we can proceed in two steps:

```
>>> elt = list[3]
>>> elt[0]
10
```

Or we can combine them:

```
>>> list[3][1]
20
```

Bracket operators evaluate from left to right, so this expression gets the three-eth element of list and extracts the one-eth element from it.

Matrices

Nested lists are often used to represent matrices. **For example**, the matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

might be represented as:

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

matrix is a list with three elements, where each element is a row of the matrix.

We can select an entire row from the matrix in the usual way:

```
>>> matrix[1]
[4, 5, 6]
```

Or we can extract a single element from the matrix using the double-index form:

```
>>> matrix[1][1]
5
```

Strings and lists

Two of the most useful functions in the string module involve lists of strings.

The **split function** breaks a string into a list of words. By default, any number of **whitespace characters** is considered a word boundary. For example:

```
>>> import string
>>> song = "The rain in Spain..."
>>> string.split(song)
['The', 'rain', 'in', 'Spain...']
```

•An optional argument called a **delimiter** can be used to specify which characters to use as word boundaries. The following example uses the string **ai** as the delimiter. For example:

```
>>> string.split(song, 'ai')
['The r', 'n in Sp', 'n...‘]
```

Notice that the delimiter doesn't appear in the list.

•The **join function** is the inverse of split. It takes a list of strings and concatenates the elements with a space between each pair. **For example:**

>>> list = ['The', 'rain', 'in', 'Spain...']

>>> “”.join(list)

'The rain in Spain...‘

•Like split, join takes an optional delimiter that is inserted between elements:

>>> “_”.join(list)

'The_rain_in_Spain...'

- 1. list.append(obj) - append obj at last
- 2. list.count(obj)- count no. of occurrence of obj in list
- 3. list.index(obj)-return index of 1st occurrence of obj
- 4. list.remove(obj)- remove 1st occurrence obj from list
- 5. a.reverse()- reverse list
- 6. a.sort()- to sort elements
- 8. min(list1)- gives min of list
- 9. max(list1)- gives max of list

- Ex: a=[10,20,30,"abc",10]
- a.append("Hi")
- a.count(10)
- a.index(10)
- a.remove(20)
- a. reverse()

- * For integer list
- a=[10,20,30]
- len(a)
- min(a)
- max(a)
- sum(a)
- a.sort()
- a.reverse()
- a.count(10)
- a.index(10)
- a.remove(10)
- a.append(10)
- a.insert(2,30)

Questions

Q1. Creation of list and changing value of any one element, also display the length of list.

Q2. Create a list and append two elements in it.

Q3. Create a list and sort it.

Q4. Create a list of numbers and print sum of all the elements.

Q5. Program to compare elements of list.

Q6. Program to find maximum and minimum of list.

Q7. Count the occurrence of element in list.

Q8. Reverse a list.

Q9.Write a loop that traverses the previous list and prints the length of each element. What happens if you send an integer to len?

- Compare two list
- Merge two list