Chapter 9
Lists

A. Downey, *Think Python: How to Think Like a Computer Science,* 3rd ed., O'Reilly, 2024.

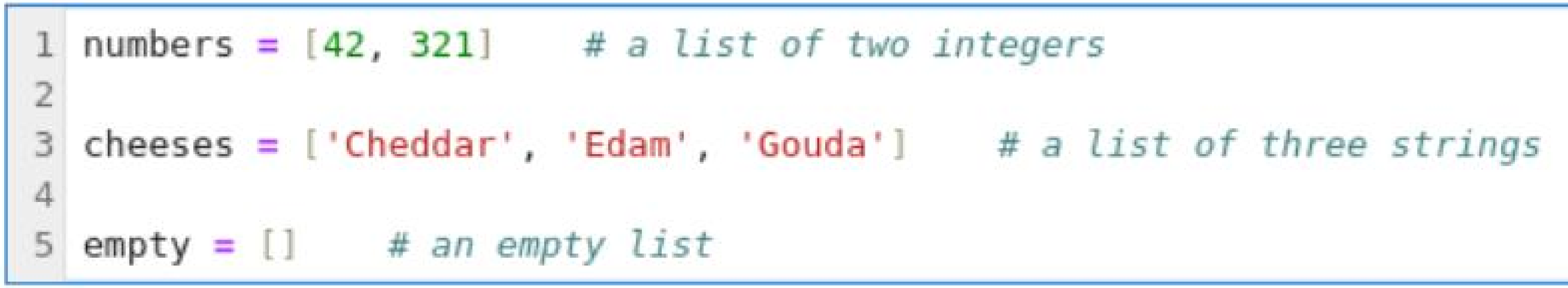
https://allendowney.github.io/ThinkPython/

Contents

Contents

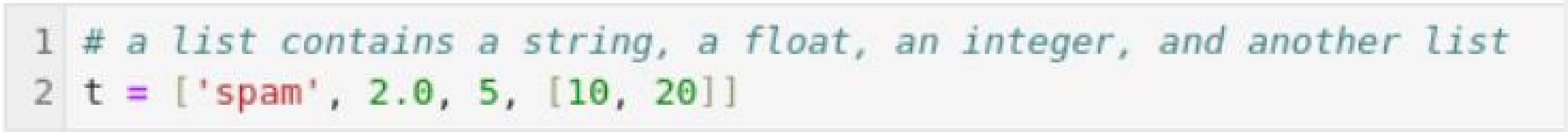
1. A List Is a Sequence

- A **list** is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called **elements**.
- There are several ways to create a new list; the simplest is to enclose the elements in square brackets (**[** and **]**).



```python
numbers = [42, 321]     # a list of two integers

cheeses = ['Cheddar', 'Edam', 'Gouda']     # a list of three strings

empty = []     # an empty list
```

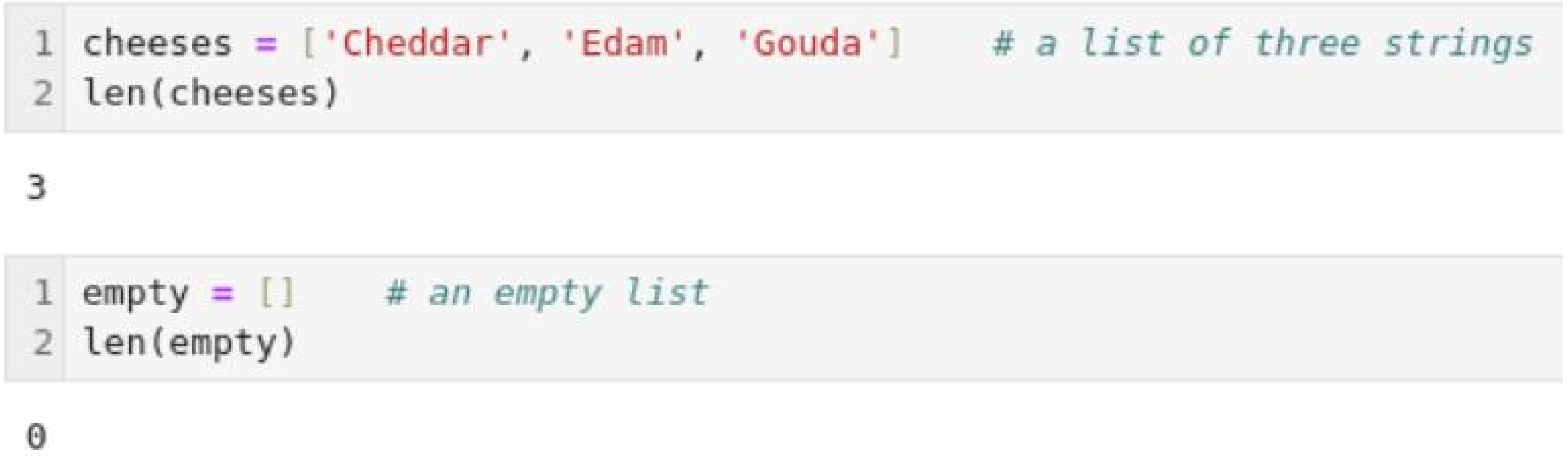
- The elements of a list don’t have to be the same type.



```python
# a list contains a string, a float, an integer, and another list
t = ['spam', 2.0, 5, [10, 20]]
```

1. A List Is a Sequence

- The **len** function returns the length of a list:



```
cheeses = ['Cheddar', 'Edam', 'Gouda']     # a list of three strings
len(cheeses)
```

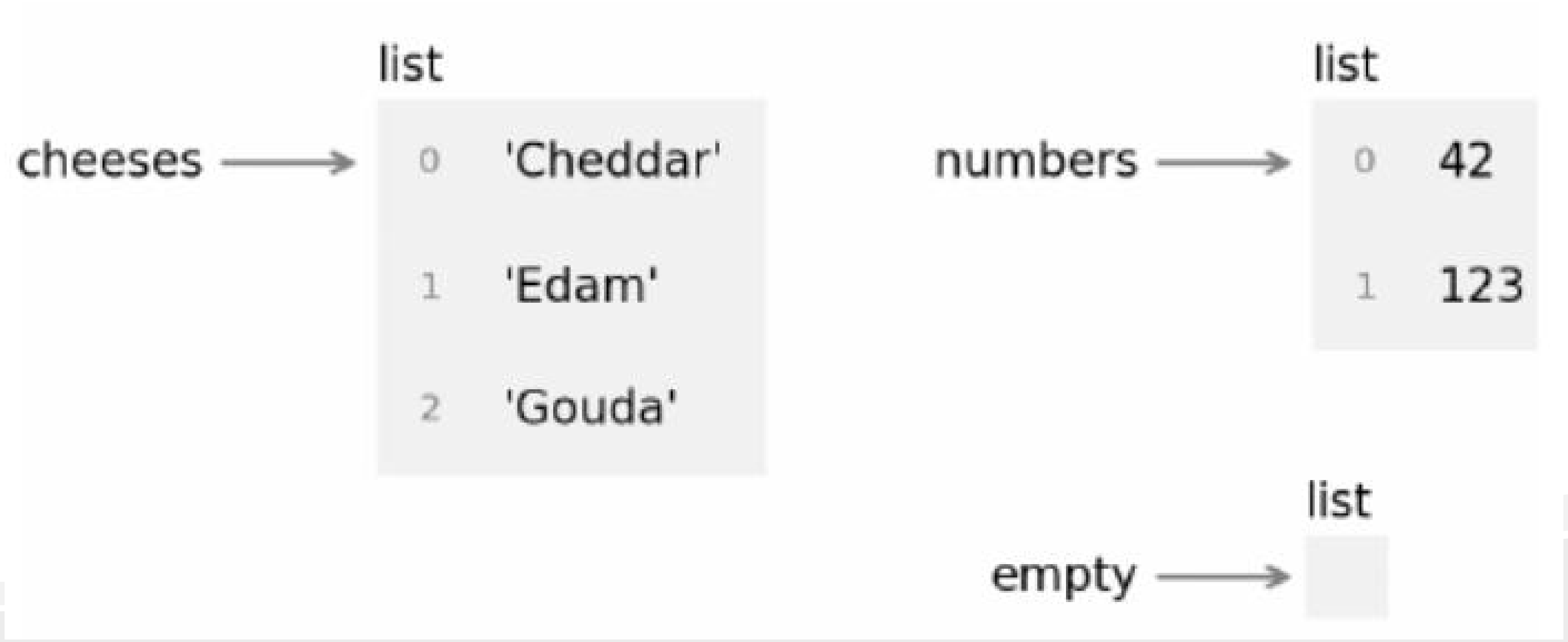
```
3
```

```
empty = []     # an empty list
len(empty)
```

```
0
```

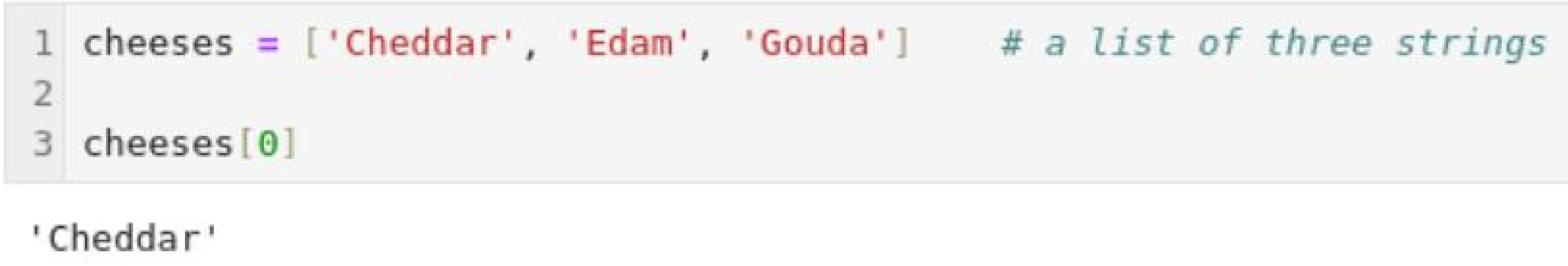
1. A List Is a Sequence

- The following figure shows the state diagram for cheeses, numbers, and empty:



2. Lists Are Mutable

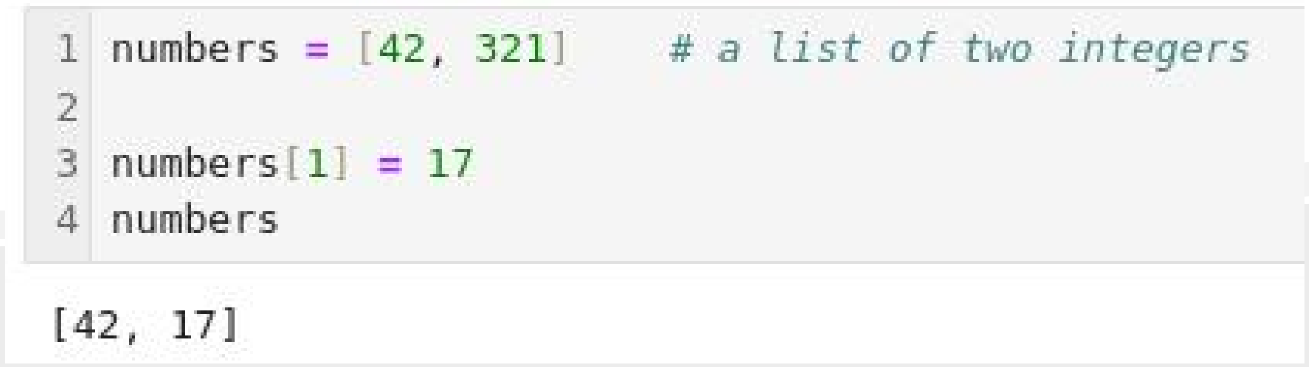
- To read an element of a list, we can use the bracket operator. The index of the first element is 0:



```
cheeses = ['Cheddar', 'Edam', 'Gouda']     # a list of three strings

cheeses[0]
```

```
'Cheddar'
```

- Unlike strings, lists are **mutable**.

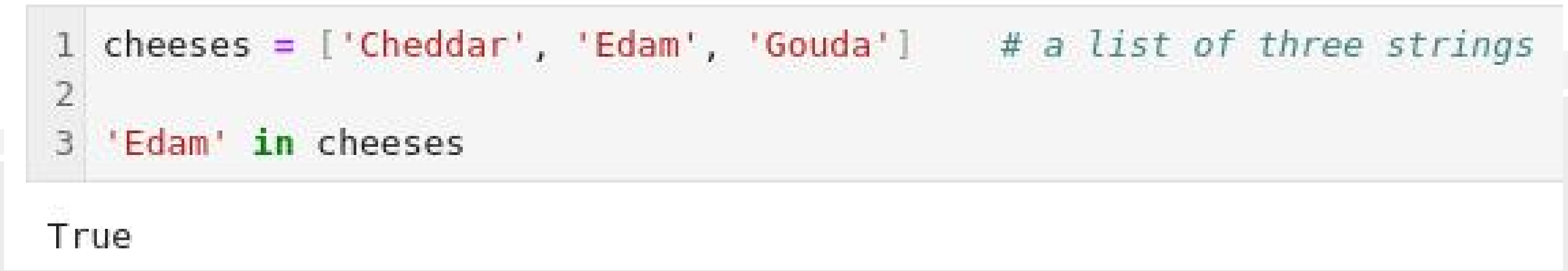


```
numbers = [42, 321]     # a list of two integers

numbers[1] = 17
numbers
```

```
[42, 17]
```

2. Lists Are Mutable

- 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.
    - The index -1 selects the last element, -2 selects the second to last, and so on.
- The **in** operator works on lists—it checks whether a given element appears anywhere in the list:

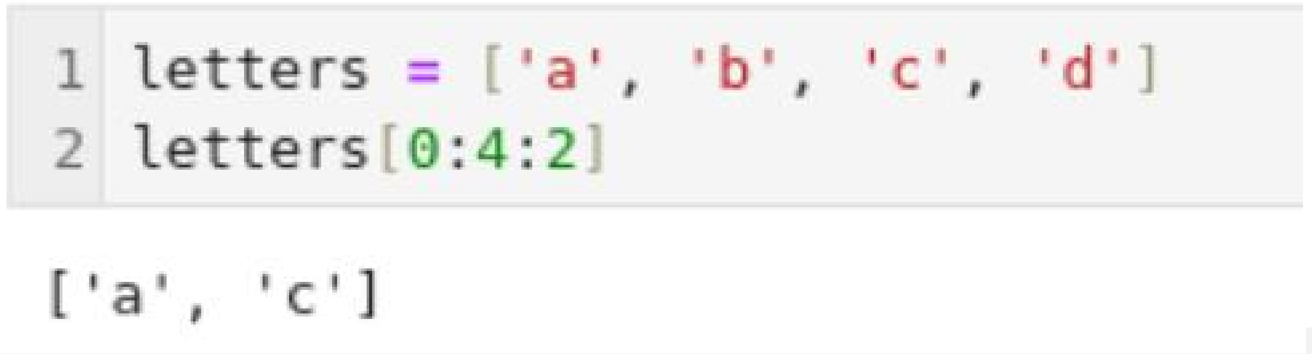


```
cheeses = ['Cheddar', 'Edam', 'Gouda']     # a list of three strings

'Edam' in cheeses
```

```
True
```

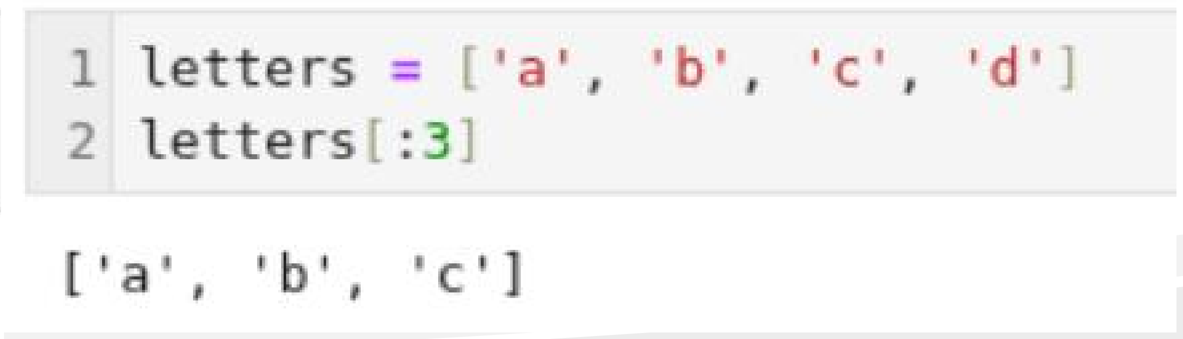
3. List Slices

- The slice operator works on lists the same way it works on strings.
- The operator [*n*:*m*:*k*] returns the part of the list from the *n*th element to the (*m* – 1)th element (inclusive). The number *k* specifies the step of the slicing (default value of *k* is 1)
  - If you omit the first index, the slice starts at the beginning of the list. If you omit the second index, the slice goes to the end of the list.



```
letters = ['a', 'b', 'c', 'd']
letters[0:4:2]
```

```
['a', 'c']
```

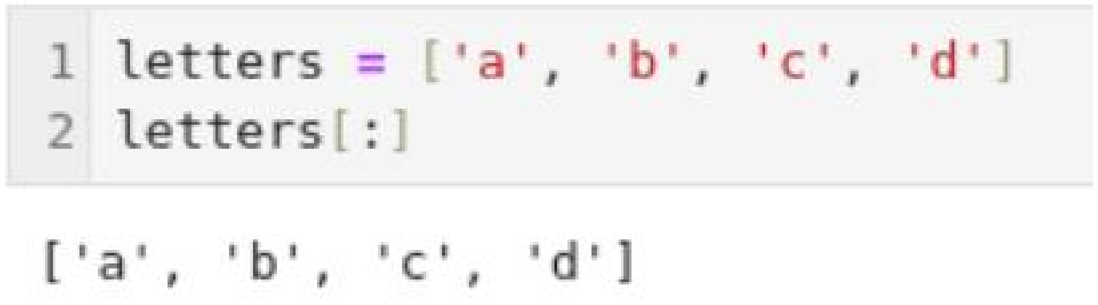


```
letters = ['a', 'b', 'c', 'd']
letters[:3]
```

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

3. List Slices

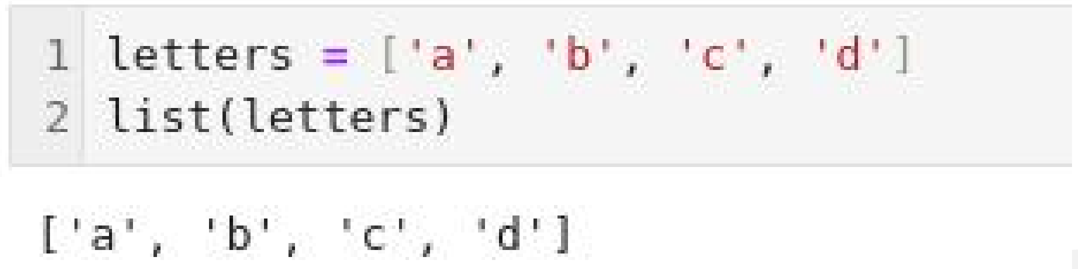
- So if you omit both, the slice is a copy of the whole list:



```
letters = ['a', 'b', 'c', 'd']
letters[:]
```

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

- Another way to copy a list is to use the **list** function:

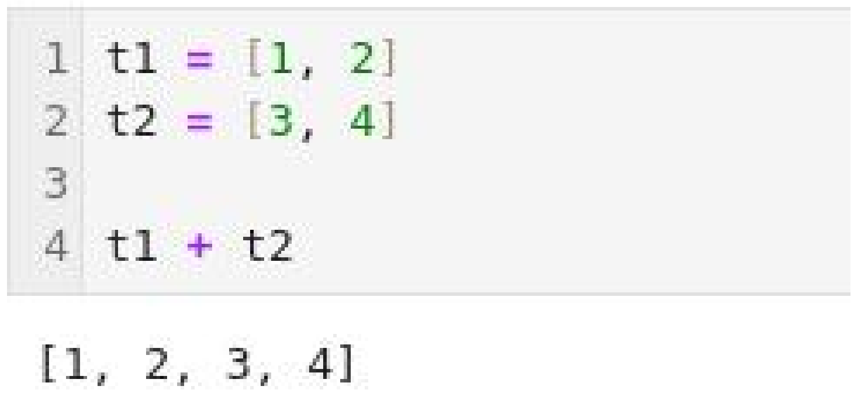


```
letters = ['a', 'b', 'c', 'd']
list(letters)
```

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

4. List Operations

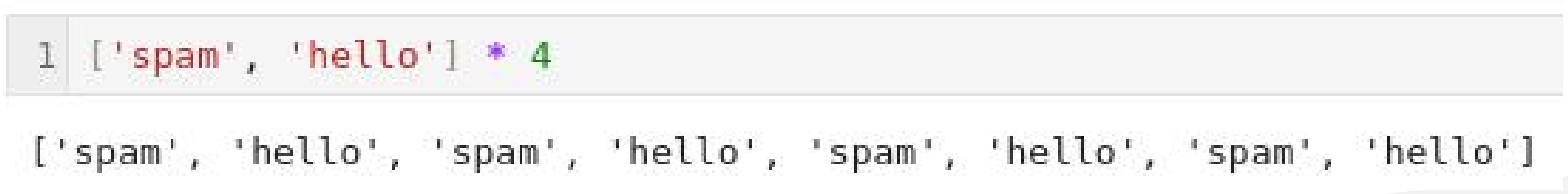
- The + operator concatenates lists:



```
t1 = [1, 2]
t2 = [3, 4]

t1 + t2
```

```
[1, 2, 3, 4]
```

- The * operator repeats a list a given number of times:

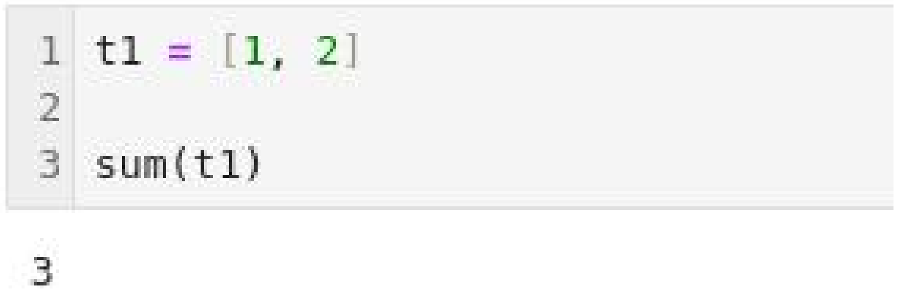


```
['spam', 'hello'] * 4
```

```
['spam', 'hello', 'spam', 'hello', 'spam', 'hello', 'spam', 'hello']
```

4. List Operations

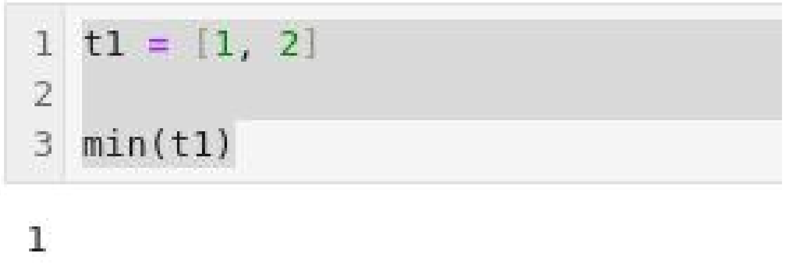
- The built-in function **sum** adds up the elements:



```
t1 = [1, 2]

sum(t1)
```

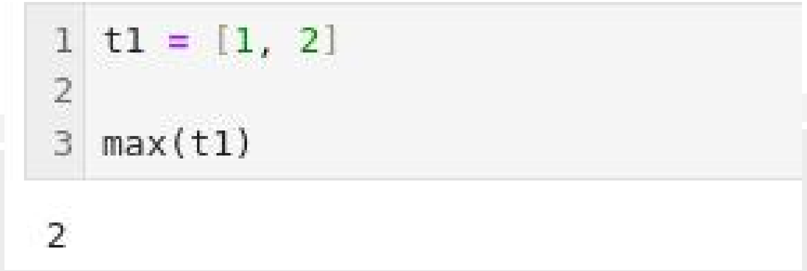
```
3
```

- **min** and **max** find the smallest and largest elements:



```
t1 = [1, 2]

min(t1)
```

```
1
```

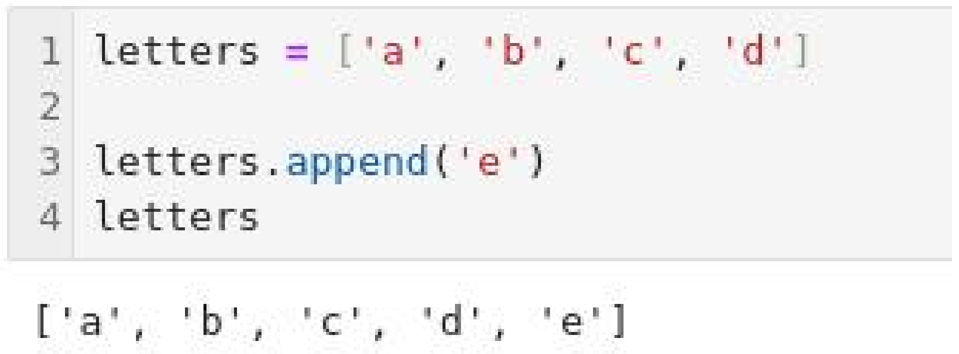


```
t1 = [1, 2]

max(t1)
```

```
2
```

5. List Methods

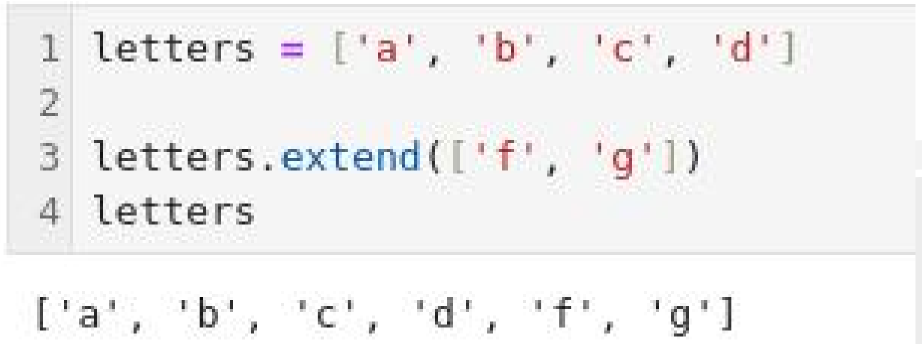
- Python provides methods that operate on lists.
- **append** adds a new element to the end of a list:



```python
letters = ['a', 'b', 'c', 'd']

letters.append('e')
letters
```

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

- **extend** takes a list as an argument and appends all of the elements:

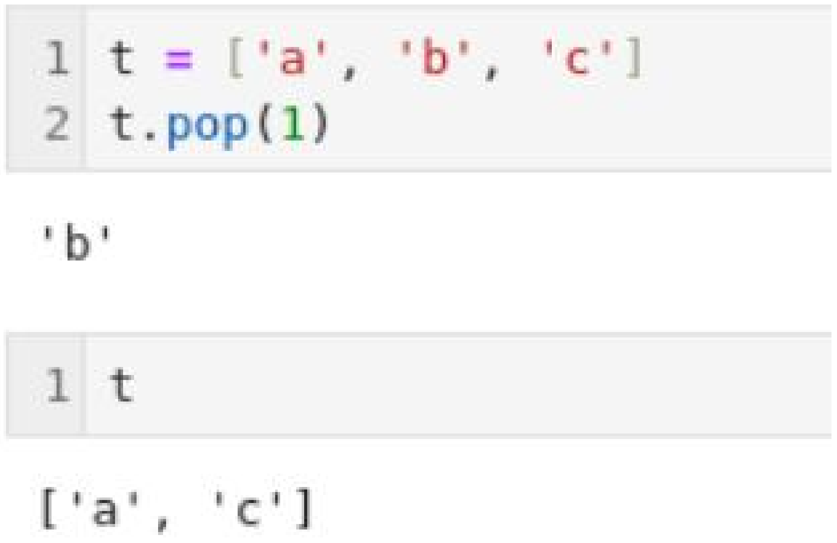


```python
letters = ['a', 'b', 'c', 'd']

letters.extend(['f', 'g'])
letters
```

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

5. List Methods

- There are two methods that remove elements from a list. If you know the index of the element you want, you can use **pop**:



```
t = ['a', 'b', 'c']
t.pop(1)
```

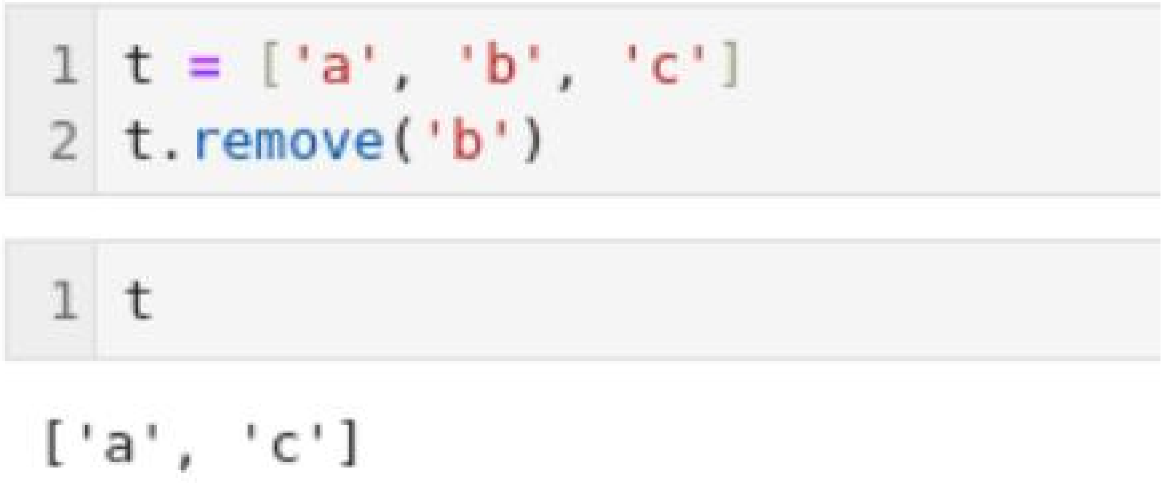
```
'b'
```

```
t
```

```
['a', 'c']
```

5. List Methods

- If you know the element you want to remove (but not the index), you can use **remove**:



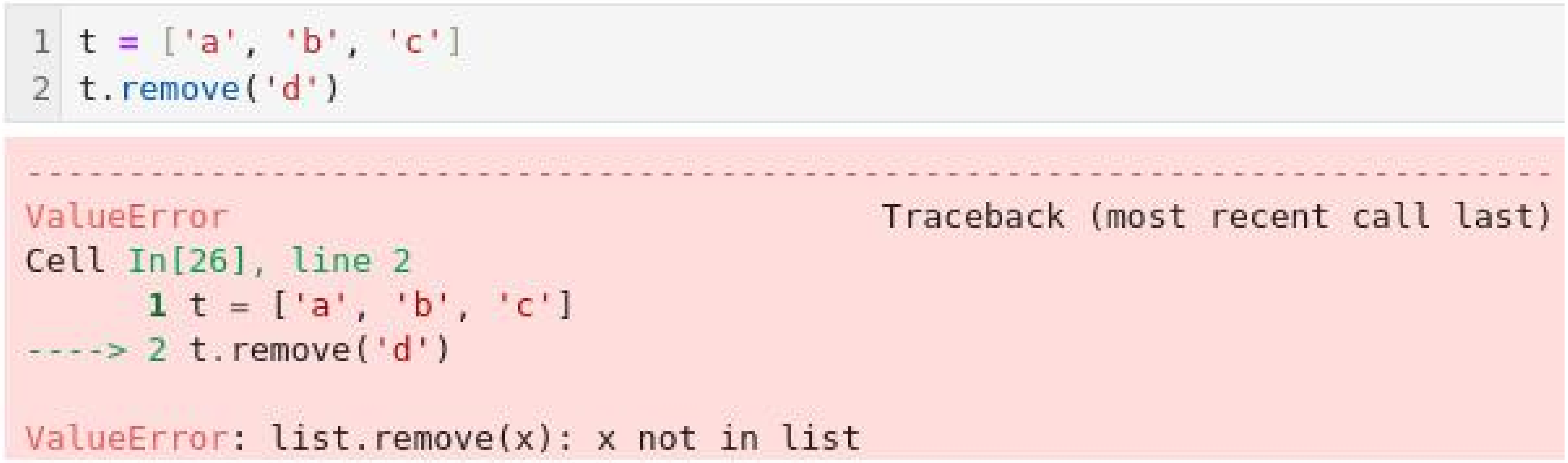
```
t = ['a', 'b', 'c']
t.remove('b')
```

```
t
```

```
['a', 'c']
```

5. List Methods

- If the element you ask for is not in the list, that’s a **ValueError**:

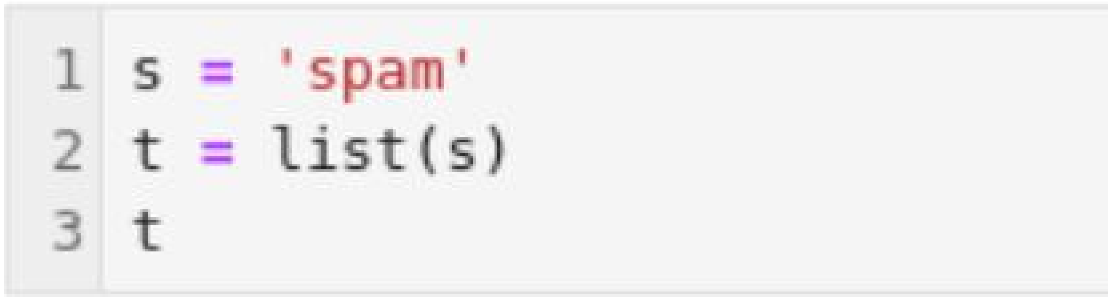


```
t = ['a', 'b', 'c']
t.remove('d')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[26], line 2
      1 t = ['a', 'b', 'c']
----> 2 t.remove('d')

ValueError: list.remove(x): x not in list
```

6. List and Strings

- A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string.
- To convert from a string to a list of characters, you can use the **list** function:

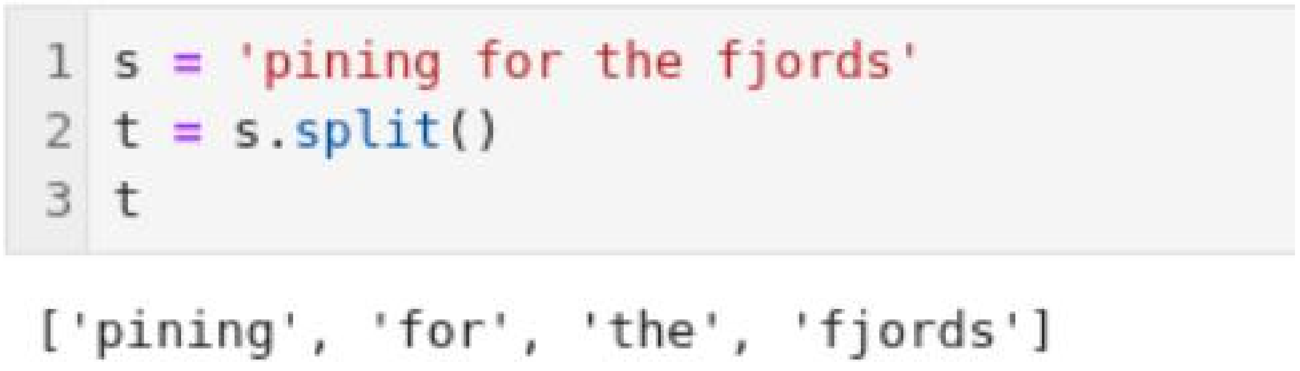


```
s = 'spam'
t = list(s)
t
```

```
['s', 'p', 'a', 'm']
```

6. List and Strings

- The list function breaks a string into individual letters. If you want to break a string into words, you can use the **split** method:

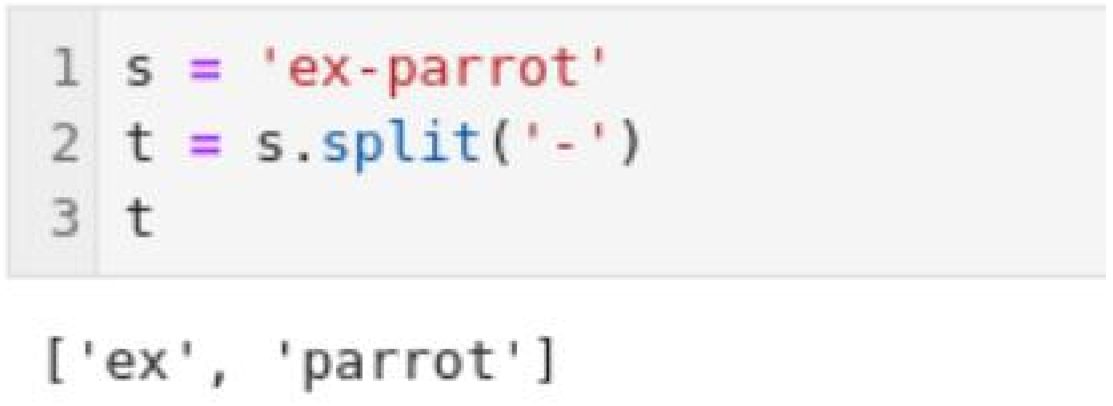


```python
s = 'pining for the fjords'
t = s.split()
t
```

```
['pining', 'for', 'the', 'fjords']
```

6. List and Strings

- The **split** method has an optional argument called a **delimiter** specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

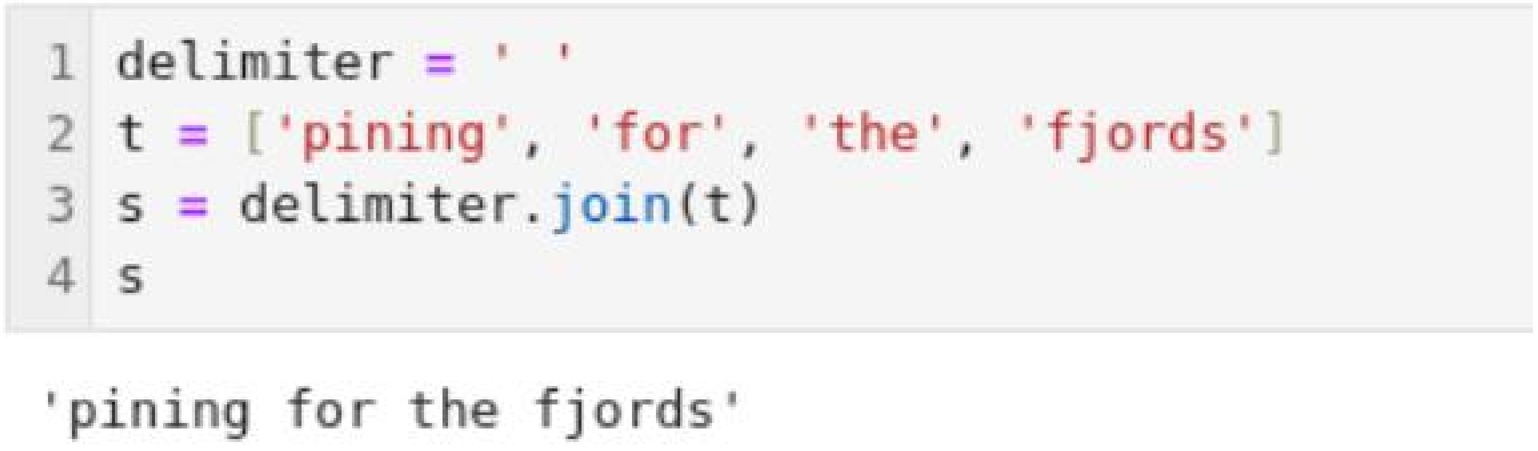


```python
s = 'ex-parrot'
t = s.split('-')
t
```

```
['ex', 'parrot']
```

6. List and Strings

- If you have a **list of strings**, you can concatenate them into a single string using **join**.
  - **join** is a **string** method, so you have to invoke it on the delimiter and pass the list as an argument:



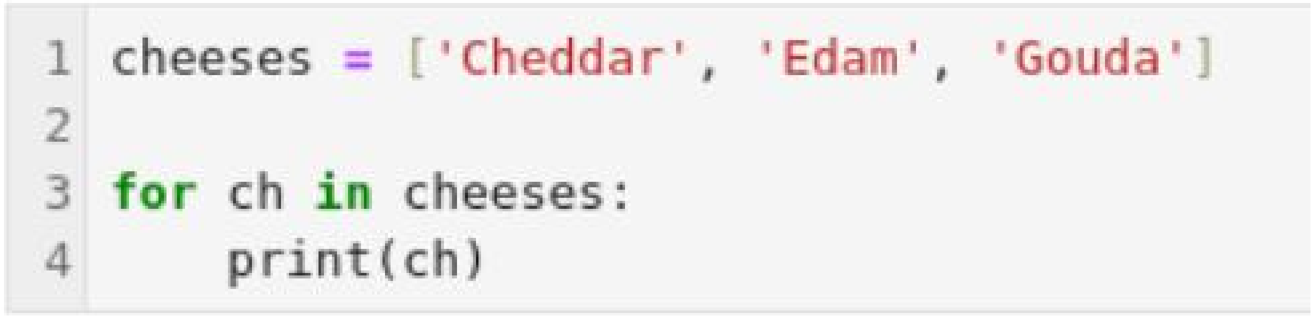
```python
delimiter = ' '
t = ['pining', 'for', 'the', 'fjords']
s = delimiter.join(t)
s
```

```
'pining for the fjords'
```

  - In this case the delimiter is a space character, so **join** puts a space between words. To join strings without spaces, you can use the empty string, '', as a delimiter.

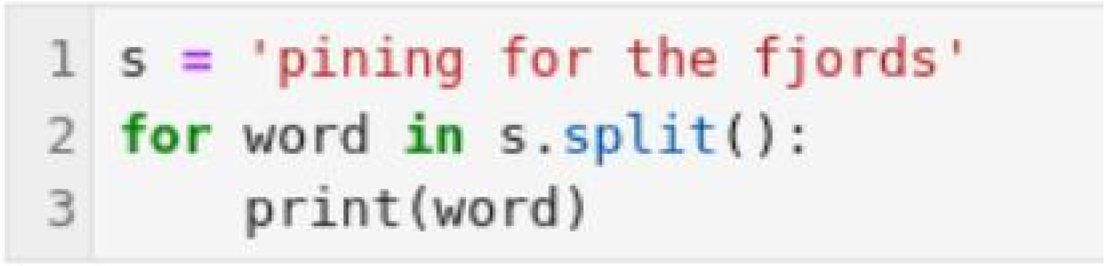
7. Looping Through a List

- You can use a **for** statement to loop through the elements of a list:



```python
cheeses = ['Cheddar', 'Edam', 'Gouda']

for ch in cheeses:
    print(ch)
```

```
Cheddar
Edam
Gouda
```

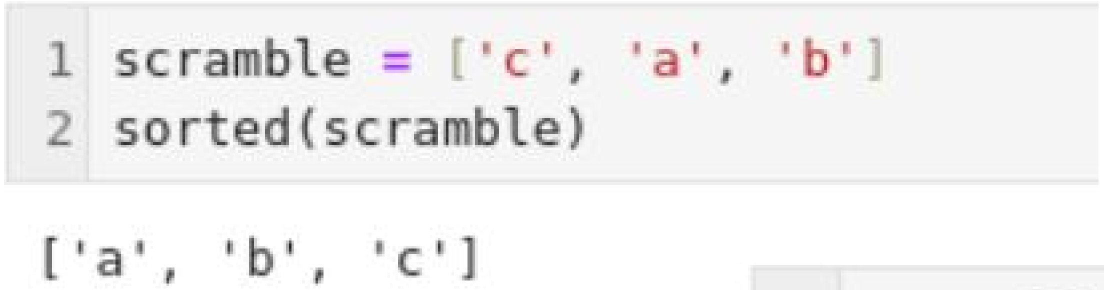


```python
s = 'pining for the fjords'
for word in s.split():
    print(word)
```

```
pining
for
the
fjords
```

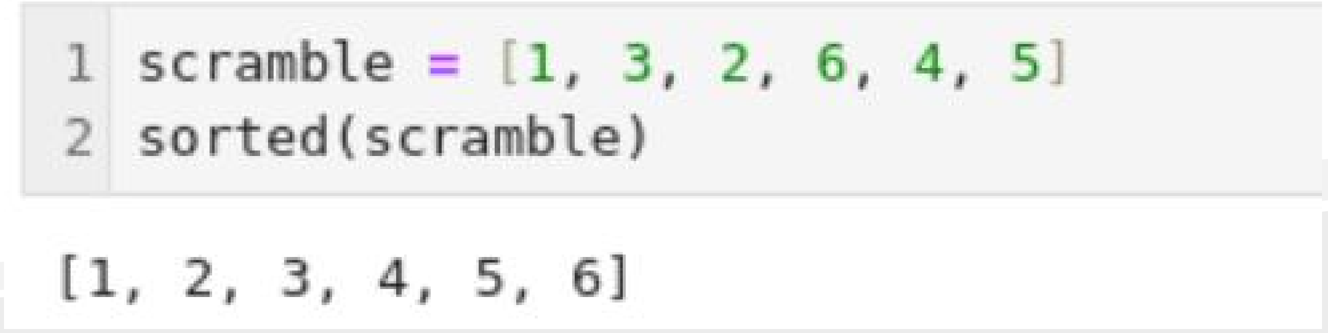
8. Sorting Lists

- Python provides a built-in function called **sorted** that sorts the elements of a list:
  - The sorted function creates a new list of sorted elements. The original list is unchanged.



```
scramble = ['c', 'a', 'b']
sorted(scramble)
```

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

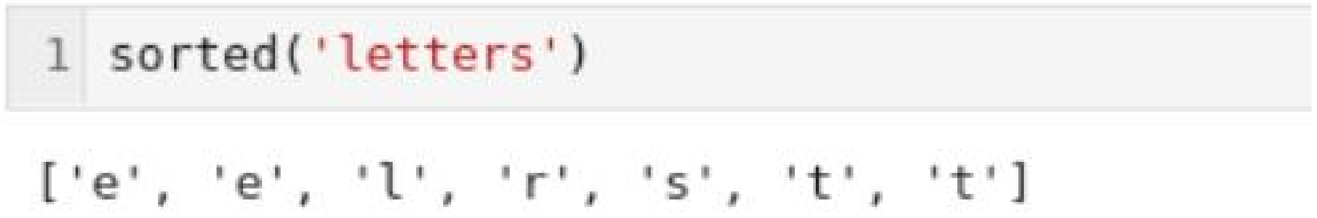


```
scramble = [1, 3, 2, 6, 4, 5]
sorted(scramble)
```

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

8. Sorting Lists

- The **sorted** function works with any kind of sequence, not just lists. So we can sort the letters in a string like this:

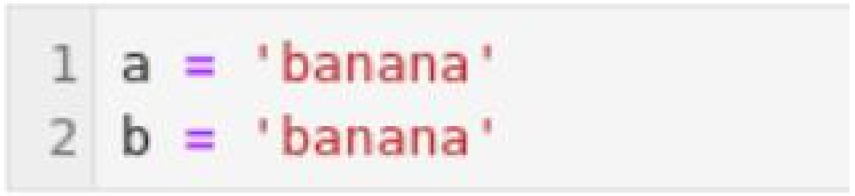


```
sorted('letters')
```

```
['e', 'e', 'l', 'r', 's', 't', 't']
```

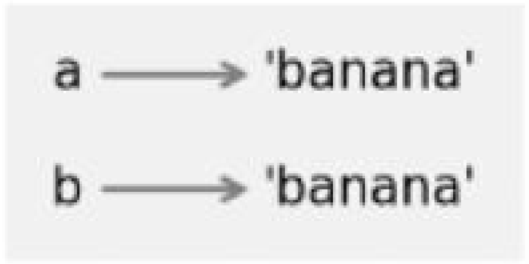
9. Objects and Values

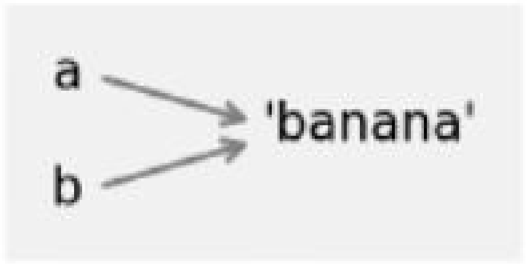
- In the following assignment statements, do **a** and **b** refer to the *same* string?



```
a = 'banana'
b = 'banana'
```

- There are two possible states, shown in the following figure:

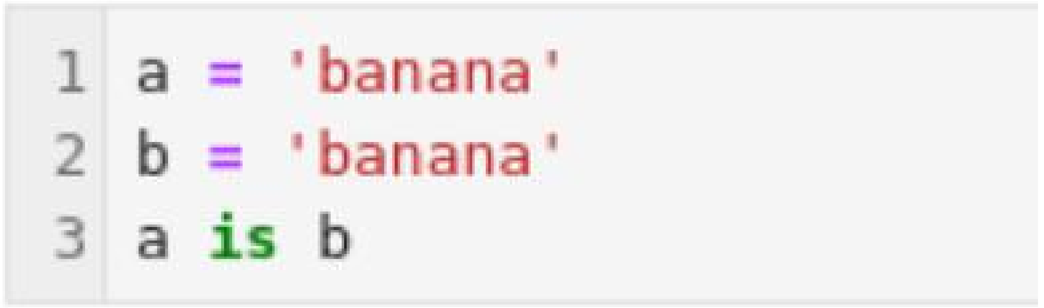




- In the diagram on the left, **a** and **b** refer to two different objects that have the same value. In the diagram on the right, they refer to the same object.

9. Objects and Values

- To check whether two variables refer to the same object, you can use the **is** operator.



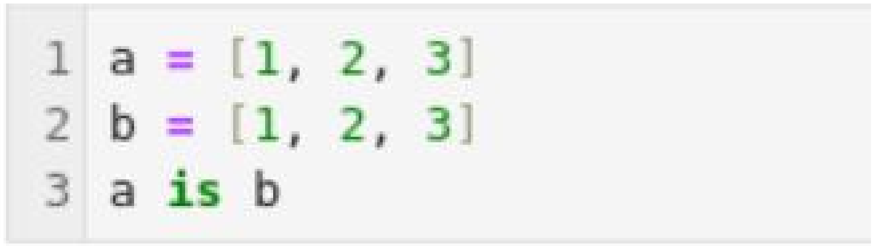
```
a = 'banana'
b = 'banana'
a is b
```

```
True
```

  - In this example, Python only created one string object, and both **a** and **b** refer to it.

9. Objects and Values

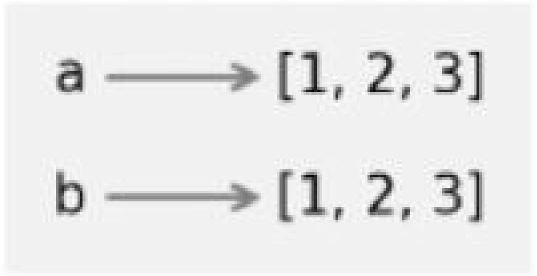
- But when you create two lists, you get two objects:



```python
a = [1, 2, 3]
b = [1, 2, 3]
a is b
```

```
False
```

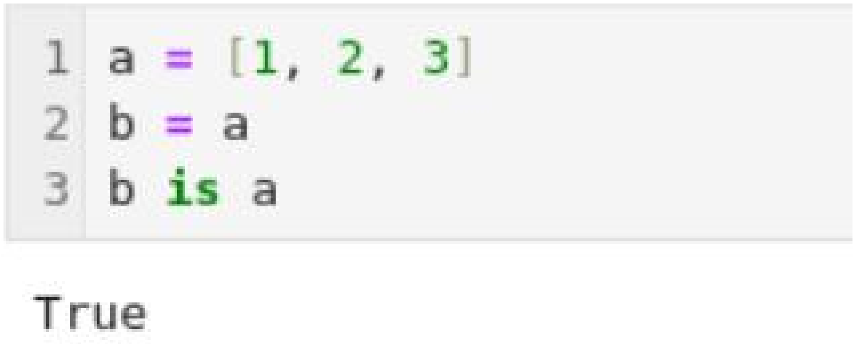
  - So the state diagram looks like this:



  - In this case we would say that the two lists are **equivalent** because they have the same elements, but are not **identical** because they are not the same object.
  - If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

10. Aliasing

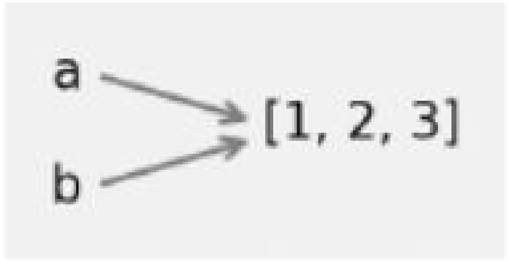
- If **a** refers to an object and you assign **b** = **a**, then both variables refer to the same object:



```
a = [1, 2, 3]
b = a
b is a
```

```
True
```

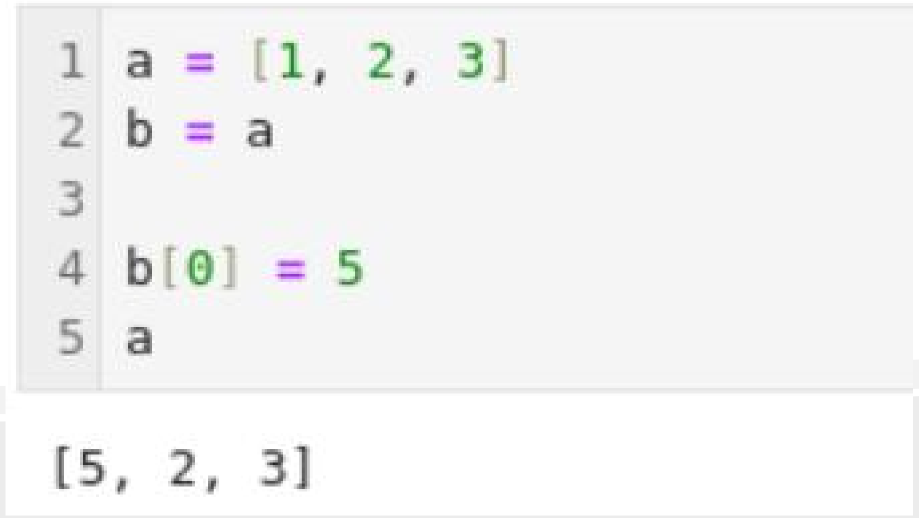
  - So the state diagram looks like this:



  - The association of a variable with an object is called a **reference**. In this example, there are two references to the same object.

10. Aliasing

- An object with more than one reference has more than one name, so we say the object is **aliased**.
- If the aliased object is **mutable**, changes made with one name affect the other.
- In this example, if we change the object **b** refers to, we are also changing the object **a** refers to (they refer to the same object):



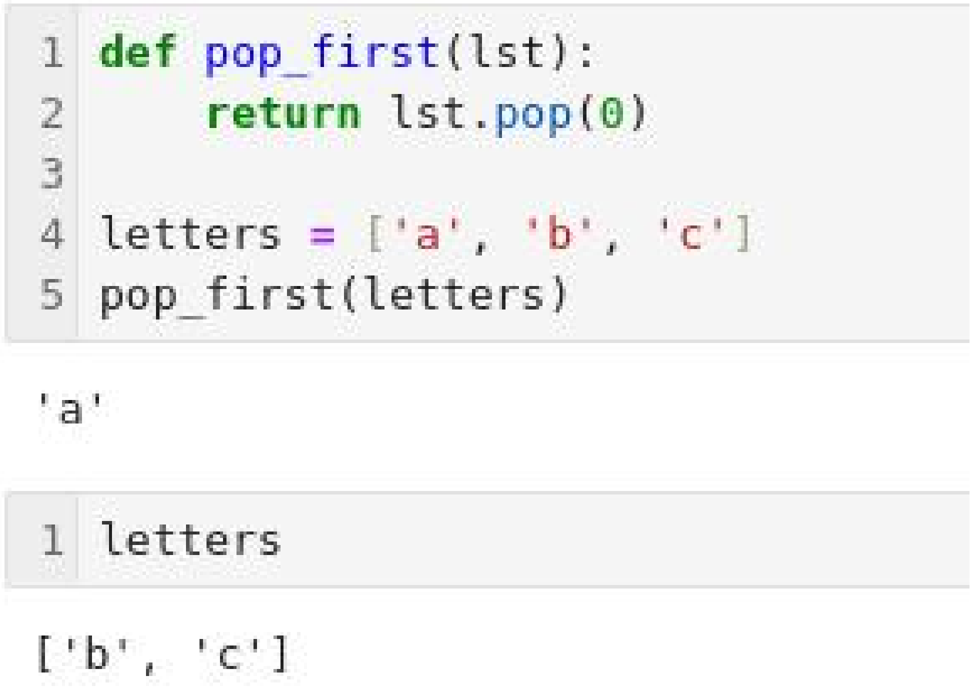
```
a = [1, 2, 3]
b = a

b[0] = 5
a
```

```
[5, 2, 3]
```

- So we would say that **a** "sees" this change. Although this behavior can be useful, it is error prone.
- In general, it is safer to avoid aliasing when you are working with **mutable** objects.

11. List Arguments

- When we pass a list to a function, the function gets a reference to the list. If the function modifies the list, the caller sees the change.

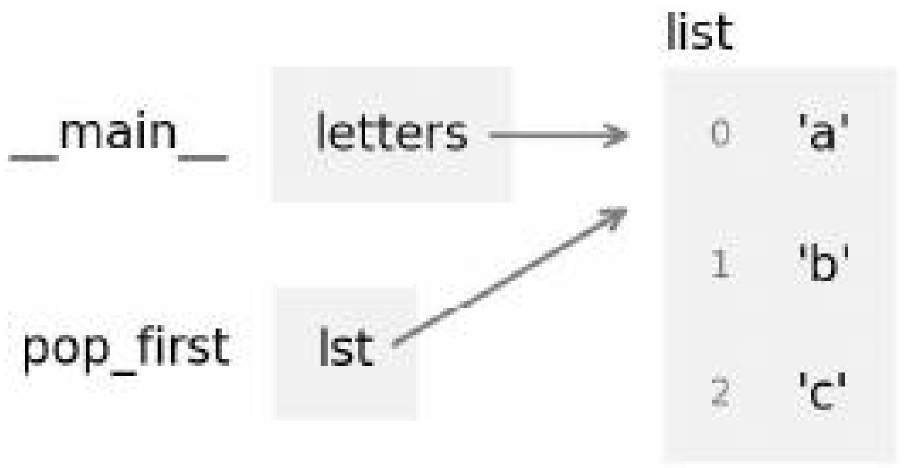


```
def pop_first(lst):
    return lst.pop(0)

letters = ['a', 'b', 'c']
pop_first(letters)
```

```
'a'
```

```
letters
```

```
['b', 'c']
```

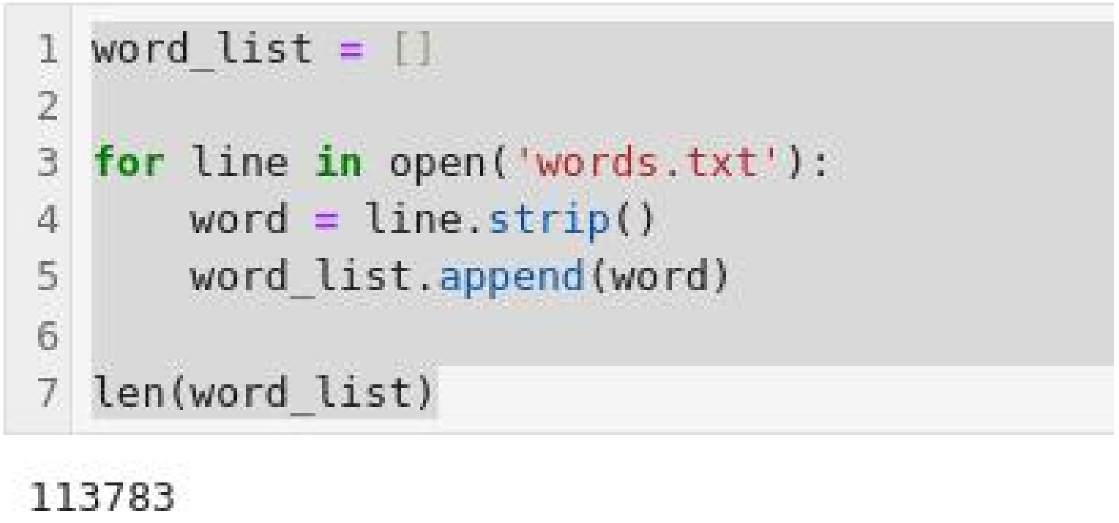


- In this example, the parameter **lst** and the variable **letters** are aliases for the same object, so the stack diagram looks like this.

- Passing a reference to an object as an argument to a function creates a form of aliasing. If the function modifies the object, those changes persist after the function is done.

12. Making a Word List

- In an application, we can read a file many times, which is not efficient. It is better to read the file once and put in a list.

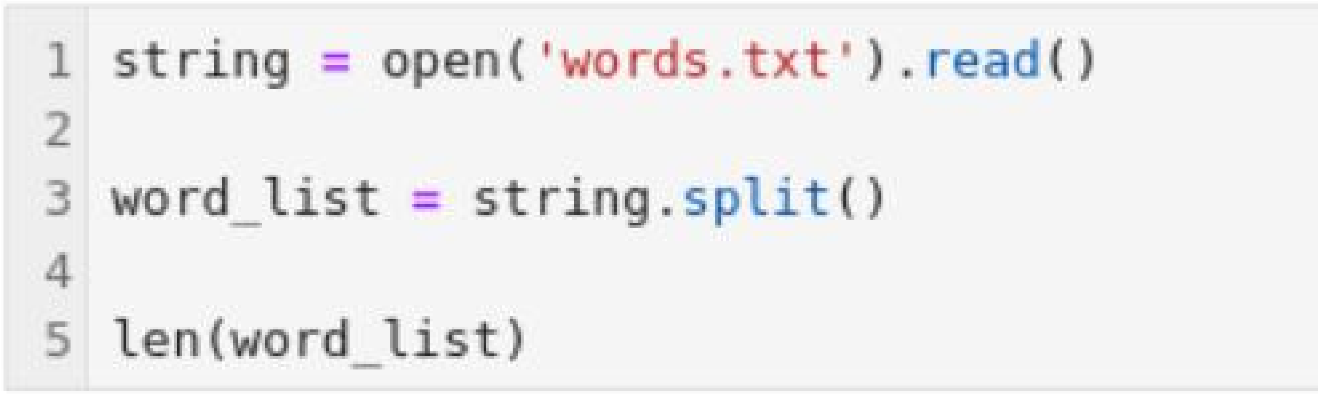


```
word_list = []

for line in open('words.txt'):
    word = line.strip()
    word_list.append(word)

len(word_list)
```

```
113783
```

12. Making a Word List

- Another way to do the same thing is to use **read** to read the entire file into a string. We can use the split method to **split** it into a list of words.

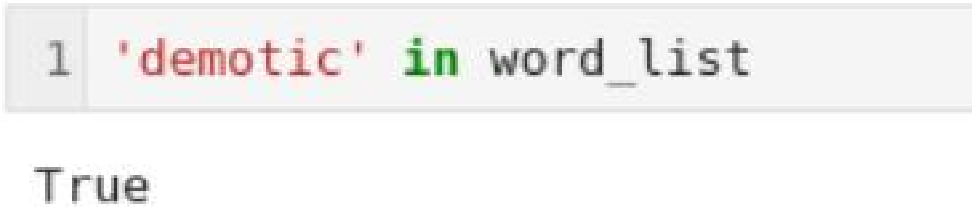


```
string = open('words.txt').read()

word_list = string.split()

len(word_list)
```

```
113783
```

12. Making a Word List

- Now, to check whether a string appears in the list, we can use the in operator. For example, 'demotic' is in the list:



```
'demotic' in word_list
```

```
True
```