

You can order print and ebook versions of *Think Python 3e* from [Bookshop.org](http://Bookshop.org) and [Amazon](http://Amazon.com).

```
In [1]: from os.path import basename, exists

def download(url):
    filename = basename(url)
    if not exists(filename):
        from urllib.request import urlretrieve

        local, _ = urlretrieve(url, filename)
        print("Downloaded " + str(local))
    return filename

download('https://github.com/AllenDowney/ThinkPython/raw/v3/thinkpython.py');
download('https://github.com/AllenDowney/ThinkPython/raw/v3/diagram.py');

import thinkpython
```

## Lists

This chapter presents one of Python's most useful built-in types, lists. You will also learn more about objects and what can happen when multiple variables refer to the same object.

In the exercises at the end of the chapter, we'll make a word list and use it to search for special words like palindromes and anagrams.

## A list is a sequence

Like a string, 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 `]` ). For example, here is a list of two integers.

```
In [2]: numbers = [42, 123]
```

And here's a list of three strings.

```
In [3]: cheeses = ['Cheddar', 'Edam', 'Gouda']
```

The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and even another list.

```
In [4]: t = ['spam', 2.0, 5, [10, 20]]
```

A list within another list is **nested**.

A list that contains no elements is called an empty list; you can create one with empty brackets, `[]`.

```
In [5]: empty = []
```

The `len` function returns the length of a list.

```
In [6]: len(cheeses)
```

```
Out[6]: 3
```

The length of an empty list is `0`.

```
In [7]: len(empty)
```

```
Out[7]: 0
```

The following figure shows the state diagram for `cheeses`, `numbers` and `empty`.

```
In [8]: from diagram import make_list, Binding, Value

list1 = make_list(cheeses, dy=-0.3, offsetx=0.17)
binding1 = Binding(Value('cheeses'), list1)

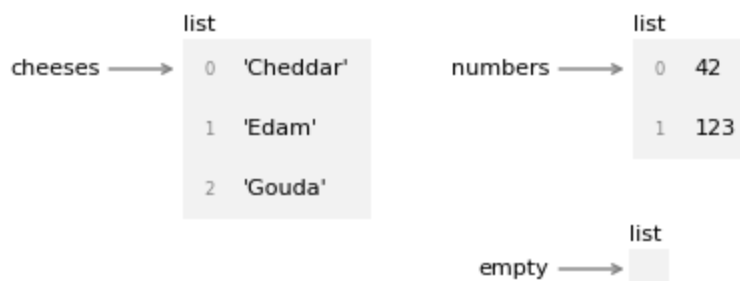
list2 = make_list(numbers, dy=-0.3, offsetx=0.17)
binding2 = Binding(Value('numbers'), list2)

list3 = make_list(empty, dy=-0.3, offsetx=0.1)
binding3 = Binding(Value('empty'), list3)
```

```
In [9]: from diagram import diagram, adjust, Bbox

width, height, x, y = [3.66, 1.58, 0.45, 1.2]
ax = diagram(width, height)
bbox1 = binding1.draw(ax, x, y)
bbox2 = binding2.draw(ax, x+2.25, y)
bbox3 = binding3.draw(ax, x+2.25, y-1.0)

bbox = Bbox.union([bbox1, bbox2, bbox3])
#adjust(x, y, bbox)
```



Lists are represented by boxes with the word "list" outside and the numbered elements of the list inside.

## Lists are mutable

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

```
In [10]: cheeses[0]
```

```
Out[10]: 'Cheddar'
```

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
In [11]: numbers[1] = 17
         numbers
```

```
Out[11]: [42, 17]
```

The second element of `numbers`, which used to be `123`, is now `17`.

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 `in` operator works on lists -- it checks whether a given element appears anywhere in the list.

```
In [12]: 'Edam' in cheeses
```

```
Out[12]: True
```

```
In [13]: 'Wensleydale' in cheeses
```

```
Out[13]: False
```

Although a list can contain another list, the nested list still counts as a single element -- so in the following list, there are only four elements.

```
In [14]: t = ['spam', 2.0, 5, [10, 20]]
         len(t)
```

```
Out[14]: 4
```

And `10` is not considered to be an element of `t` because it is an element of a nested list, not `t`.

```
In [15]: 10 in t
```

```
Out[15]: False
```

## List slices

The slice operator works on lists the same way it works on strings. The following example selects the second and third elements from a list of four letters.

```
In [16]: letters = ['a', 'b', 'c', 'd']  
letters[1:3]
```

```
Out[16]: ['b', 'c']
```

If you omit the first index, the slice starts at the beginning.

```
In [17]: letters[:2]
```

```
Out[17]: ['a', 'b']
```

If you omit the second, the slice goes to the end.

```
In [18]: letters[2:]
```

```
Out[18]: ['c', 'd']
```

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

```
In [19]: letters[:]
```

```
Out[19]: ['a', 'b', 'c', 'd']
```

Another way to copy a list is to use the `list` function.

```
In [20]: list(letters)
```

```
Out[20]: ['a', 'b', 'c', 'd']
```

Because `list` is the name of a built-in function, you should avoid using it as a variable name.

## List operations

The `+` operator concatenates lists.

```
In [21]: t1 = [1, 2]
         t2 = [3, 4]
         t1 + t2
```

```
Out[21]: [1, 2, 3, 4]
```

The `*` operator repeats a list a given number of times.

```
In [22]: ['spam'] * 4
```

```
Out[22]: ['spam', 'spam', 'spam', 'spam']
```

No other mathematical operators work with lists, but the built-in function `sum` adds up the elements.

```
In [23]: sum(t1)
```

```
Out[23]: 3
```

And `min` and `max` find the smallest and largest elements.

```
In [24]: min(t1)
```

```
Out[24]: 1
```

```
In [25]: max(t2)
```

```
Out[25]: 4
```

## List methods

Python provides methods that operate on lists. For example, `append` adds a new element to the end of a list:

```
In [26]: letters.append('e')
         letters
```

```
Out[26]: ['a', 'b', 'c', 'd', 'e']
```

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

```
In [27]: letters.extend(['f', 'g'])
         letters
```

```
Out[27]: ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

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

```
In [28]: t = ['a', 'b', 'c']  
t.pop(1)
```

```
Out[28]: 'b'
```

The return value is the element that was removed. And we can confirm that the list has been modified.

```
In [29]: t
```

```
Out[29]: ['a', 'c']
```

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

```
In [30]: t = ['a', 'b', 'c']  
t.remove('b')
```

The return value from `remove` is `None` . But we can confirm that the list has been modified.

```
In [31]: t
```

```
Out[31]: ['a', 'c']
```

If the element you ask for is not in the list, that's a `ValueError`.

```
In [32]: t.remove('d')
```

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

## Lists 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.

```
In [33]: s = 'spam'  
t = list(s)  
t
```

```
Out[33]: ['s', 'p', 'a', 'm']
```

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

```
In [34]: s = 'pining for the fjords'  
t = s.split()  
t
```

```
Out[34]: ['pining', 'for', 'the', 'fjords']
```

An optional argument called a **delimiter** specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter.

```
In [35]: s = 'ex-parrot'
         t = s.split('-')
         t
```

```
Out[35]: ['ex', 'parrot']
```

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.

```
In [36]: delimiter = ' '
         t = ['pining', 'for', 'the', 'fjords']
         s = delimiter.join(t)
         s
```

```
Out[36]: '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.

## Looping through a list

You can use a `for` statement to loop through the elements of a list.

```
In [37]: for cheese in cheeses:
         print(cheese)
```

```
Cheddar
Edam
Gouda
```

For example, after using `split` to make a list of words, we can use `for` to loop through them.

```
In [38]: s = 'pining for the fjords'

         for word in s.split():
             print(word)
```

```
pining
for
the
fjords
```

A `for` loop over an empty list never runs the indented statements.

```
In [39]: for x in []:
```

```
print('This never happens.')
```

## Sorting lists

Python provides a built-in function called `sorted` that sorts the elements of a list.

```
In [40]: scramble = ['c', 'a', 'b']
sorted(scramble)
```

```
Out[40]: ['a', 'b', 'c']
```

The original list is unchanged.

```
In [41]: scramble
```

```
Out[41]: ['c', 'a', 'b']
```

`sorted` works with any kind of sequence, not just lists. So we can sort the letters in a string like this.

```
In [42]: sorted('letters')
```

```
Out[42]: ['e', 'e', 'l', 'r', 's', 't', 't']
```

The result is a list. To convert the list to a string, we can use `join`.

```
In [43]: ''.join(sorted('letters'))
```

```
Out[43]: 'eelrstt'
```

With an empty string as the delimiter, the elements of the list are joined with nothing between them.

## Objects and values

If we run these assignment statements:

```
In [44]: a = 'banana'
b = 'banana'
```

We know that `a` and `b` both refer to a string, but we don't know whether they refer to the *same* string. There are two possible states, shown in the following figure.

```
In [45]: from diagram import Frame, Stack

s = 'banana'
bindings = [Binding(Value(name), Value(repr(s))) for name in 'ab']
frame1 = Frame(bindings, dy=-0.25)
```



```

binding1 = Binding(Value('a'), Value(repr(s)), dy=-0.11)
binding2 = Binding(Value('b'), draw_value=False, dy=0.11)
frame2 = Frame([binding1, binding2], dy=-0.25)

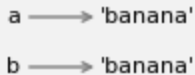
stack = Stack([frame1, frame2], dx=1.7, dy=0)

```

```

In [46]: width, height, x, y = [2.85, 0.76, 0.17, 0.51]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)
# adjust(x, y, bbox)

```



```

a → 'banana'
b → 'banana'

```



```

a → 'banana'
b → 'banana'

```

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

```

In [47]: a = 'banana'
b = 'banana'
a is b

```

Out[47]: True

In this example, Python only created one string object, and both `a` and `b` refer to it. But when you create two lists, you get two objects.

```

In [48]: a = [1, 2, 3]
b = [1, 2, 3]
a is b

```

Out[48]: False

So the state diagram looks like this.

```

In [49]: t = [1, 2, 3]
binding1 = Binding(Value('a'), Value(repr(t)))
binding2 = Binding(Value('b'), Value(repr(t)))
frame = Frame([binding1, binding2], dy=-0.25)

```

```

In [50]: width, height, x, y = [1.16, 0.76, 0.21, 0.51]
ax = diagram(width, height)
bbox = frame.draw(ax, x, y)
# adjust(x, y, bbox)

```

```
a → [1, 2, 3]
b → [1, 2, 3]
```

In this case we would say that the two lists are **equivalent**, because they have the same elements, but 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.

## Aliasing

If `a` refers to an object and you assign `b = a`, then both variables refer to the same object.

```
In [51]: a = [1, 2, 3]
         b = a
         b is a
```

```
Out[51]: True
```

So the state diagram looks like this.

```
In [52]: t = [1, 2, 3]
         binding1 = Binding(Value('a'), Value(repr(t)), dy=-0.11)
         binding2 = Binding(Value('b'), draw_value=False, dy=0.11)
         frame = Frame([binding1, binding2], dy=-0.25)
```

```
In [53]: width, height, x, y = [1.11, 0.81, 0.17, 0.56]
         ax = diagram(width, height)
         bbox = frame.draw(ax, x, y)
         # adjust(x, y, bbox)
```

```
a → [1, 2, 3]
b → [1, 2, 3]
```

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

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.

```
In [54]: b[0] = 5
         a
```

```
Out[54]: [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.

For immutable objects like strings, aliasing is not as much of a problem. In this example:

```
In [55]: a = 'banana'
         b = 'banana'
```

It almost never makes a difference whether `a` and `b` refer to the same string or not.

## List arguments

When you 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. For example, `pop_first` uses the list method `pop` to remove the first element from a list.

```
In [56]: def pop_first(lst):
         return lst.pop(0)
```

We can use it like this.

```
In [57]: letters = ['a', 'b', 'c']
         pop_first(letters)
```

```
Out[57]: 'a'
```

The return value is the first element, which has been removed from the list -- as we can see by displaying the modified list.

```
In [58]: letters
```

```
Out[58]: ['b', 'c']
```

In this example, the parameter `lst` and the variable `letters` are aliases for the same object, so the state diagram looks like this:

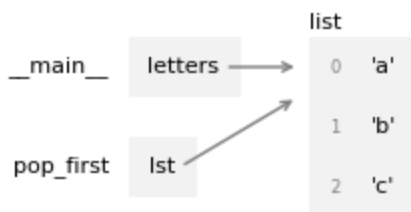
```
In [59]: lst = make_list('abc', dy=-0.3, offsetx=0.1)
binding1 = Binding(Value('letters'), draw_value=False)
frame1 = Frame([binding1], name='__main__', loc='left')

binding2 = Binding(Value('lst'), draw_value=False, dx=0.61, dy=0.35)
frame2 = Frame([binding2], name='pop_first', loc='left', offsetx=0.08)

stack = Stack([frame1, frame2], dx=-0.3, dy=-0.5)
```

```
In [60]: width, height, x, y = [2.04, 1.24, 1.06, 0.85]
ax = diagram(width, height)
bbox1 = stack.draw(ax, x, y)
bbox2 = lst.draw(ax, x+0.5, y)
bbox = Bbox.union([bbox1, bbox2])
adjust(x, y, bbox)
```

Out[60]: [2.05, 1.22, 1.06, 0.85]



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.

(section\_word\_list)=

## Making a word list

In the previous chapter, we read the file `words.txt` and searched for words with certain properties, like using the letter `e`. But we read the entire file many times, which is not efficient. It is better to read the file once and put the words in a list. The following loop shows how.

```
In [61]: download('https://raw.githubusercontent.com/AllenDowney/ThinkPython/v3/words.txt');
```

```
In [62]: word_list = []

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

len(word_list)
```

Out[62]: 113783

Before the loop, `word_list` is initialized with an empty list. Each time through the loop, the `append` method adds a word to the end. When the loop is done, there are more than 113,000 words in the list.

Another way to do the same thing is to use `read` to read the entire file into a string.

```
In [63]: string = open('words.txt').read()
len(string)
```

```
Out[63]: 1016511
```

The result is a single string with more than a million characters. We can use the `split` method to split it into a list of words.

```
In [64]: word_list = string.split()
len(word_list)
```

```
Out[64]: 113783
```

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

```
In [65]: 'demotic' in word_list
```

```
Out[65]: True
```

But `'contrafibularities'` is not.

```
In [66]: 'contrafibularities' in word_list
```

```
Out[66]: False
```

And I have to say, I'm anaspeptic about it.

## Debugging

Note that most list methods modify the argument and return `None`. This is the opposite of the string methods, which return a new string and leave the original alone.

If you are used to writing string code like this:

```
In [67]: word = 'plumage!'
word = word.strip('!')
word
```

```
Out[67]: 'plumage'
```

It is tempting to write list code like this:

```
In [68]: t = [1, 2, 3]
        t = t.remove(3)           # WRONG!
```

`remove` modifies the list and returns `None`, so next operation you perform with `t` is likely to fail.

```
In [69]: t.remove(2)
```

```
AttributeError: 'NoneType' object has no attribute 'remove'
```

This error message takes some explaining. An **attribute** of an object is a variable or method associated with it. In this case, the value of `t` is `None`, which is a `NoneType` object, which does not have a attribute named `remove`, so the result is an `AttributeError`.

If you see an error message like this, you should look backward through the program and see if you might have called a list method incorrectly.

## Glossary

**list:** An object that contains a sequence of values.

**element:** One of the values in a list or other sequence.

**nested list:** A list that is an element of another list.

**delimiter:** A character or string used to indicate where a string should be split.

**equivalent:** Having the same value.

**identical:** Being the same object (which implies equivalence).

**reference:** The association between a variable and its value.

**aliased:** If there is more than one variable that refers to an object, the object is aliased.

**attribute:** One of the named values associated with an object.

## Exercises

```
In [70]: # This cell tells Jupyter to provide detailed debugging information
        # when a runtime error occurs. Run it before working on the exercises.

        %xmode Verbose
```

Exception reporting mode: Verbose

## Ask a virtual assistant

In this chapter, I used the words "contrafibularities" and "anaspeptic", but they are not actually English words. They were used in the British television show *Black Adder*, Season 3, Episode 2, "Ink and Incapability".

However, when I asked ChatGPT 3.5 (August 3, 2023 version) where those words came from, it initially claimed they are from Monty Python, and later claimed they are from the Tom Stoppard play *Rosencrantz and Guildenstern Are Dead*.

If you ask now, you might get different results. But this example is a reminder that virtual assistants are not always accurate, so you should check whether the results are correct. As you gain experience, you will get a sense of which questions virtual assistants can answer reliably. In this example, a conventional web search can identify the source of these words quickly.

If you get stuck on any of the exercises in this chapter, consider asking a virtual assistant for help. If you get a result that uses features we haven't learned yet, you can assign the VA a "role".

For example, before you ask a question try typing "Role: Basic Python Programming Instructor". After that, the responses you get should use only basic features. If you still see features we you haven't learned, you can follow up with "Can you write that using only basic Python features?"

*I don't understand what 'doing' this particular exercise actually entails, so I am skipping it.*

## Exercise

Two words are anagrams if you can rearrange the letters from one to spell the other. For example, `tops` is an anagram of `stop`.

One way to check whether two words are anagrams is to sort the letters in both words. If the lists of sorted letters are the same, the words are anagrams.

Write a function called `is_anagram` that takes two strings and returns `True` if they are anagrams.

To get you started, here's an outline of the function with doctests.

```
In [71]: def is_anagram(word1, word2):
          """Checks whether two words are anagrams.

          >>> is_anagram('tops', 'stop')
          True
          >>> is_anagram('skate', 'takes')
          True
          >>> is_anagram('tops', 'takes')
          False
          >>> is_anagram('skate', 'stop')
```

```
False
"""
return sorted(word1) == sorted(word2)
```

You can use `doctest` to test your function.

```
In [72]: from doctest import run_docstring_examples

def run_doctests(func):
    run_docstring_examples(func, globals(), name=func.__name__)

run_doctests(is_anagram)
```

Using your function and the word list, find all the anagrams of `takes`.

```
In [73]: for word in word_list:
        if is_anagram(word, "takes"):
            print(word)
```

```
skate
stake
steak
takes
teaks
```

## Exercise

Python provides a built-in function called `reversed` that takes as an argument a sequence of elements -- like a list or string -- and returns a `reversed` object that contains the elements in reverse order.

```
In [74]: reversed('parrot')
```

```
Out[74]: <reversed at 0x19f3ba6bfd0>
```

If you want the reversed elements in a list, you can use the `list` function.

```
In [75]: list(reversed('parrot'))
```

```
Out[75]: ['t', 'o', 'r', 'r', 'a', 'p']
```

Of if you want them in a string, you can use the `join` method.

```
In [76]: ''.join(reversed('parrot'))
```

```
Out[76]: 'torrap'
```

So we can write a function that reverses a word like this.

```
In [77]: def reverse_word(word):
        return ''.join(reversed(word))
```



A palindrome is a word that is spelled the same backward and forward, like "noon" and "rotator". Write a function called `is_palindrome` that takes a string argument and returns `True` if it is a palindrome and `False` otherwise.

Here's an outline of the function with doctests you can use to check your function.

```
In [78]: def is_palindrome(word):
        """Check if a word is a palindrome.

        >>> is_palindrome('bob')
        True
        >>> is_palindrome('alice')
        False
        >>> is_palindrome('a')
        True
        >>> is_palindrome('')
        True
        """
        return word == reverse_word(word)
```

```
In [79]: run_doctests(is_palindrome)
```

You can use the following loop to find all of the palindromes in the word list with at least 7 letters.

```
In [80]: for word in word_list:
        if len(word) >= 7 and is_palindrome(word):
            print(word)
```

```
deified
halalah
reifier
repaper
reviver
rotator
sememes
```

## Exercise

Write a function called `reverse_sentence` that takes as an argument a string that contains any number of words separated by spaces. It should return a new string that contains the same words in reverse order. For example, if the argument is "Reverse this sentence", the result should be "Sentence this reverse".

Hint: You can use the `capitalize` methods to capitalize the first word and convert the other words to lowercase.

To get you started, here's an outline of the function with doctests.

```
In [83]: def reverse_sentence(input_string):
        '''Reverse the words in a string and capitalize the first.

        >>> reverse_sentence('Reverse this sentence')
        'Sentence this reverse'

        >>> reverse_sentence('Python')
        'Python'

        >>> reverse_sentence('')
        ''

        >>> reverse_sentence('One for all and all for one')
        'One for all and all for one'
        '''
        sentence = ' '.join(reversed(input_string.split()))
        sentence = sentence.capitalize()
        return sentence
```

```
In [84]:
```

```
In [84]: run_doctests(reverse_sentence)
```

## Exercise

Write a function called `total_length` that takes a list of strings and returns the total length of the strings. The total length of the words in `word_list` should be 902728.

```
In [85]: def total_length(string_list):
        total = 0
        for string in string_list:
            total += len(string)
        return total
```

```
In [86]: total_length(word_list)
```

```
Out[86]: 902728
```