Chapter 8

Strings

Strings are not like integers, floats, and booleans. A string is a **sequence (序列)**, which means it is an ordered collection of other values. In this chapter you'll see how to access the characters that make up a string, and you'll learn about some of the methods strings provide.

8.1 A string is a sequence

A string is a sequence of characters.
You can access the characters one at a time with the bracket operator:

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

The second statement selects character number 1 from `fruit` and assigns it to `letter`.

The expression in brackets is called an **index (索引)**.
The index indicates which character in the sequence you want (hence the name).

But you might not get what you expect:

```
>>> letter
'a'
```

For most people, the first letter of 'banana' is `b`, not `a`. But for computer scientists, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
>>> letter = fruit[0]
>>> letter
'b'
```

So `b` is the 0th letter ("zero-eth") of 'banana', `a` is the 1th letter ("one-eth"), and `n` is the 2th letter ("two-eth").

As an index you can use an expression that contains variables and operators:

```
>>> i = 1
>>> fruit[i]
'a'
>>> fruit[i+1]
'n'
```

But the value of the index has to be an integer. Otherwise you get:

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

In [2]:
```
fruit = 'banana'
letter = fruit[1]
```

In [3]:
```
i = 1
print(letter)
print(fruit[0])
print(fruit[i+1])
```

```
a
b
n
```

In [5]:
```
print(letter[1.4])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-e3feea6438f8> in <module>()
----> 1 print(letter[1.4])

TypeError: string indices must be integers
```

8.2 len

`len` is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

The reason for the `IndexError` is that there is no letter in `'banana'` with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from `length` :

```
>>> last = fruit[length-1]
>>> last
'a'
```

Or you can use negative indices, which count backward from the end of the string. The expression `fruit[-1]` yields the last letter, `fruit[-2]` yields the second to last, and so on.

In [6]:
```
print(len(fruit))
```

```
6
```

In [7]:
```
length = len(fruit)
print(length)
print(fruit[length - 1])
print(fruit[length])
```

```
6
a
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-7-4179ef9dcf45> in <module>()
      2 print(length)
      3 print(fruit[length - 1])
----> 4 print(fruit[length])

IndexError: string index out of range
```

In [8]:
```
print(fruit[-1], fruit[-2])
```

```
a
n
```

8.3 Traversal (遍歷) with a for loop

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal (遍歷)**. One way to write a traversal is with a `while` loop:

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is `index < len(fruit)`, so when `index` is equal to the length of the string, the condition is false, and the body of the loop doesn't run. The last character accessed is the one with the index `len(fruit)-1`, which is the last character in the string.

As an exercise, write a function that takes a string as an argument and displays the letters backward, one per line.

Another way to write a traversal is with a `for` loop:

```
for letter in fruit:
    print(letter)
```

Each time through the loop, the next character in the string is assigned to the variable `letter`. The loop continues until no characters are left.

The following example shows how to use concatenation (string addition) and a `for` loop to generate an abecedarian series (that is, in alphabetical order). In Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'

for letter in prefixes:
    print(letter + suffix)
```

The output is:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Of course, that's not quite right because "Ouack" and "Quack" are misspelled. As an exercise, modify the program to fix this error.

從頭開始逐一找出每一個字母，並且列印

In [1]:
```
fruit = 'banana'
```

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

```
b
a
n
a
n
a
```

從尾端開始逐一找出每一個字母，並且列印

In [3]:

```
# write a function that takes a string as an argument and displays the letters
# backward, one per line.
fruit = 'banana'
index = 1
while index <= len(fruit):
    letter = fruit[-index]
    print(letter)
    index = index + 1
```

```
a
n
a
n
a
b
```

In [4]:

```
x = range(6)
s = "banana"
print(type(x))
print(type(s))
print(x)
print(s)
```

```
<class 'range'>
<class 'str'>
range(0, 6)
banana
```

以 **for loop** 遍歷一個字串

In [5]:

```
fruit = 'banana'
for i in range(len(fruit)):
    print(fruit[i])
```

```
b
a
n
a
n
a
```

在上述範例中，range(len(fruit)) = range(6)，因此在

```
for i in range(len(fruit)):
```

的迴圈中，i 的值依序為 0,1,2,3,4,5，

```
print(fruit[i])
```

將依序列印出每一個字母。

In [15]:

```
for letter in fruit:
    print(letter)
```

```
b
a
n
a
n
a
```

上述程式示範

```
for ... in ...
```

的另一種用法，在

```
for letter in fruit:
```

的迴圈中，letter 的值依序被指派為 fruit 的每一個字母，

```
print(letter)
```

將依序列印出每一個字母。

以 for loop 連接兩個字串

In [22]:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'
for letter in prefixes:
    print(letter + suffix)
```

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

In [23]:

```
prefixes = 'JKLMNOPQ'
suffix = 'ack'
for letter in prefixes:
    if letter == 'O' or letter == 'Q':
        print(letter + 'u' + suffix)
    else:
        print(letter + suffix)
```

```
Jack
Kack
Lack
Mack
Nack
Ouack
Pack
Quack
```

8.4 String slices

A segment of a string is called a **slice**. Selecting a slice is similar to selecting a character:

```
>>> s = 'Monty Python'
>>> s[0:5]
'Monty'
>>> s[6:12]
'Python'
```

The operator `[n:m]` returns the part of the string from the "n-eth" character to the "m-eth" character, including the first but excluding the last. This behavior is counterintuitive, but it might help to imagine the indices pointing *between* the characters, as in Figure 8.1.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:</font>

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

If the first index is greater than or equal to the second the result is an **empty string**, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

Continuing this example, what do you think `fruit[:]` means? Try it and see.

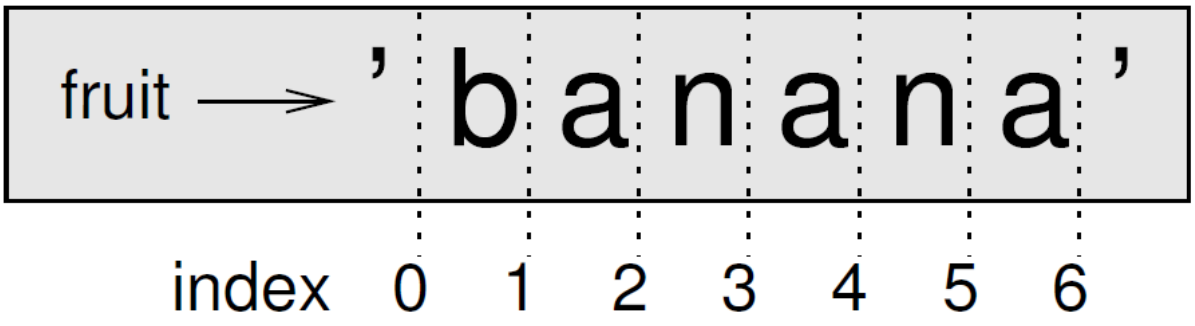


Figure 8.1: Slice indices.

```
In [7]:
s = 'Monty Python'
print(s)
print(s[6:12])
print(s[6:13])
print('lenght of s = ', len(s))
print(s[len(s)])
```

```
Monty Python
Python
Python
lenght of s =  12
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-7-d26324759667> in <module>
      4 print(s[6:13])
      5 print('lenght of s = ', len(s))
----> 6 print(s[len(s)])

IndexError: string index out of range
```

```
In [8]:
fruit = 'banana'
fruit[3:3]
```

```
Out[8]: ''
```

```
In [9]:
fruit = 'banana'
print(fruit[:3])
print(fruit[3:])
print(fruit[:])
```

```
ban
ana
banana
```

8.5 Strings are immutable (字串是不可變的)

It is tempting to use the `[]` operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

The "object" in this case is the string and the "item" is the character you tried to assign. For now, an object is the same thing as a value, but we will refine that definition later (Section 10.10).

The reason for the error is that strings are **immutable (不可變的)**, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

This example concatenates a new first letter onto a slice of `greeting`. It has no effect on the original string.

```
In [10]:
name = 'Peter Pan'
name[0] = 'K'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-10-e5e971d472f2> in <module>
      1 name = 'Peter Pan'
----> 2 name[0] = 'K'

TypeError: 'str' object does not support item assignment
```

```
In [11]:
name = "Peter Pan"
name = name[:5] + ' Rabbit'
print(name)
```

```
Peter Rabbit
```

8.6 Searching (搜尋)

What does the following function do?

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
```

In a sense, `find` is the inverse of the `[]` operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns `-1`.

This is the first example we have seen of a `return` statement inside a loop. If `word[index] == letter`, the function breaks out of the loop and returns immediately.

If the character doesn't appear in the string, the program exits the loop normally and returns `-1`.

This pattern of computation---traversing a sequence and returning when we find what we are looking for---is called a **search**.

As an exercise, modify `find` so that it has a third parameter, the index in `word` where it should start looking.

In [2]:

```python
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1

fruit = 'Apple'
print(find(fruit, 'b'))
print(find(fruit, 'A'))
print(find(fruit, 'e'))
print(find(fruit, 'c'))
```

```
-1
0
4
-1
```

In [6]:

```python
def find(word, letter):
    index = 0
    while index < len(word):
        print('index = {}'.format(index))
        if word[index] == letter:
            return index
        index = index + 1

    return -1

fruit = 'Apple'
print(find(fruit, 'b'))
print(find(fruit, 'A'))
print(find(fruit, 'e'))
print(find(fruit, 'c'))
```

```
index = 0
index = 1
index = 2
index = 3
index = 4
-1
index = 0
0
index = 0
index = 1
index = 2
index = 3
index = 4
4
index = 0
index = 1
index = 2
index = 3
index = 4
-1
```

In [30]:

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1

fruit = 'banana'
s = 'abcde'

for ch in s:
    i = find(fruit, ch)
    if i == -1:
        print(ch, ' is not in ', fruit)
    else:
        print(ch, ' is found at index ', i)
```

```
a  is found at index  1
b  is found at index  0
c  is not in  banana
d  is not in  banana
e  is not in  banana
```

In [31]:

```
def find2(word, letter, start):
    index = start
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
print(find2(fruit, 'b', 3))
print(find2(fruit, 'a', 3))
print(find2(fruit, 'c', 3))
```

```
-1
3
-1
```

8.7 Looping and counting

The following program counts the number of times the letter `a` appears in a string:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

This program demonstrates another pattern of computation called a **counter**. The variable `count` is initialized to 0 and then incremented each time an `a` is found. When the loop exits, `count` contains the result---the total number of $a's$.

As an exercise, encapsulate this code in a function named `count` , and generalize it so that it accepts the string and the letter as arguments.

Then rewrite the function so that instead of traversing the string, it uses the three-parameter version of `find` from the previous section.

In [12]:

```
word = 'banana'
count = 0
```

```
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

```
3
```

In [22]:

```
def count(word, letter):
    n = 0
    for ch in word:
        if letter == ch:
            n = n + 1
    print(n)

fruit = 'banana'
count(fruit,'a')
count(fruit,'b')
count(fruit,'c')
count('Apple', 'l')
```

```
3
1
0
1
```

8.8 String methods

Strings provide methods that perform a variety of useful operations. A method is similar to a function---it takes arguments and returns a value---but the syntax is different. For example, the method `upper` takes a string and returns a new string with all uppercase letters.

Instead of the function syntax `upper(word)` , it uses the method syntax `word.upper()` .

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> new_word
'BANANA'
```

This form of dot notation specifies the name of the method, `upper` , and the name of the string to apply the method to, `word` . The empty parentheses indicate that this method takes no arguments.

A method call is called an **invocation (調用)**; in this case, we would say that we are invoking $upper$ on $w \text{ or } d$.

As it turns out, there is a string method named `find` that is remarkably similar to the function we wrote:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> index
1
```

In this example, we invoke `find` on `word` and pass the letter we are looking for as a parameter.

Actually, the `find` method is more general than our function; it can find substrings, not just characters:

```
>>> word.find('na')
2
```

By default, `find` starts at the beginning of the string, but it can take a second argument, the index where it should start:

```
>>> word.find('na', 3)
4
```

This is an example of an **optional argument (任意引數)**; `find` can also take a third argument, the index where it should stop:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

This search fails because `b` does not appear in the index range from `1` to `2`, not including `2`. Searching up to, but not including, the second index makes `find` consistent with the slice operator.

In [15]:

```
fruit = 'banana'
s = fruit.upper()
print(s)
```

```
BANANA
```

In [16]:

```
print(s.lower())
```

```
banana
```

In [17]:

```
print(fruit.find('a'))
```

```
1
```

In [18]:

```
print(fruit.find('c'))
```

```
-1
```

In [19]:

```
word = 'banana'

index = word.find('na')
print(index)
index = word.find('Na')
print(index)
```

```
2
-1
```

In [20]:

```
name = 'bob'
name.find('b', 1, 2)
```

Out[20]:

```
-1
```

8.9 The in operator

The word `in` is a boolean operator that takes two strings and returns `True` if the first appears as a substring in the second:

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

For example, the following function prints all the letters from `word1` that also appear in `word2`:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

With well-chosen variable names, Python sometimes reads like English. You could read this loop, "for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter."

Here's what you get if you compare apples and oranges:

```
>>> in_both('apples', 'oranges')
a
e
s
```

In [35]:
```
'a' in 'banana'
```
Out[35]:
```
True
```

In [36]:
```
'seed' in 'banana'
```
Out[36]:
```
False
```

請仔細研究以下三個版本的 in_both() 之差異

In [37]:
```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)

in_both('apples', 'oranges')
```
```
a
e
s
```

In [38]:
```
def in_both(word1, word2):
    count = 0
    for letter in word1:
        if letter in word2:
            count += 1
            print(letter)
    return count

r = in_both('apples', 'oranges')
print("number of charaters found: ", r)
```
```
a
e
s
number of charaters found:  3
```

In [39]:
```
def in_both(word1, word2):
    result = ''
    for letter in word1:
        if letter in word2:
            #result = result + letter
            result += letter
            print(letter)
    return result

r = in_both('apples', 'oranges')
print("charaters found: ", r)
```
```
a
e
```

```
s
charaters found:  aes
```

8.10 String comparison

The relational operators work on strings. To see if two strings are equal:

```
if word == 'banana':
    print('All right, bananas.')
```

Other relational operations are useful for putting words in alphabetical order:

```
if word < 'banana':
    print('Your word, ' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word, ' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

Python does not handle uppercase and lowercase letters the same way people do. All the uppercase letters come before all the lowercase letters, so:

```
Your word, Pineapple, comes before banana.
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. Keep that in mind in case you have to defend yourself against a man armed with a Pineapple.

In [40]:
```
if word == 'banana':
    print('All right, bananas.')
```

```
All right, bananas.
```

In [41]:
```
if word < 'banana':
    print('Your word, ' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word, ' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

```
All right, bananas.
```

In [42]:
```
word = 'Pineapple'

if word < 'banana':
    print('Your word, ' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word, ' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

```
Your word, Pineapple, comes before banana.
```

8.11 Debugging

When you use indices to traverse the values in a sequence, it is tricky to get the beginning and end of the traversal right. Here is a function that is supposed to compare two words and return `True` if one of the words is the reverse of the other, but it contains two errors:

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False

    i = 0
    j = len(word2)

    while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1

    return True
```

The first `if` statement checks whether the words are the same length. If not, we can return `False` immediately. Otherwise, for the rest of the function, we can assume that the words are the same length. This is an example of the guardian pattern in Section 6.8.

`i` and `j` are indices: `i` traverses `word1` forward while `j` traverses `word2` backward. If we find two letters that don't match, we can return `False` immediately. If we get through the whole loop and all the letters match, we return `True` .

If we test this function with the words "pots" and "stop", we expect the return value `True` , but we get an IndexError:

```
>>> is_reverse('pots', 'stop')
...
  File "reverse.py", line 15, in is_reverse
    if word1[i] != word2[j]:
IndexError: string index out of range
```

For debugging this kind of error, my first move is to print the values of the indices immediately before the line where the error appears.

```
    while j > 0:
        print(i, j)        # print here

        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1
```

Now when I run the program again, I get more information:

```
>>> is_reverse('pots', 'stop')
0 4
...
IndexError: string index out of range
```

The first time through the loop, the value of `j` is 4, which is out of range for the string 'pots'. The index of the last character is 3, so the initial value for `j` should be `len(word2)-1` .

If I fix that error and run the program again, I get:

```
>>> is_reverse('pots', 'stop')
0 3
1 2
```

```
2 1
True
```

This time we get the right answer, but it looks like the loop only ran three times, which is suspicious. To get a better idea of what is happening, it is useful to draw a state diagram. During the first iteration, the frame for `is_reverse` is shown in Figure 8.2.

I took some license by arranging the variables in the frame and adding dotted lines to show that the values of `i` and `j` indicate characters in `word1` and `word2` .

Starting with this diagram, run the program on paper, changing the values of `i` and `j` during each iteration. Find and fix the second error in this function.

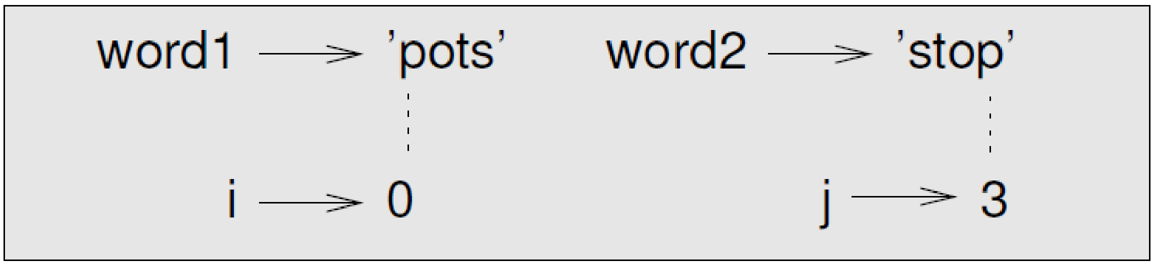


Figure 8.2: State diagram.

8.12 Glossary

- **object (物件):** Something a variable can refer to. For now, you can use "object" and "value" interchangeably.
- **sequence (序列):** An ordered collection of values where each value is identified by an integer index.
- **item (項目):** One of the values in a sequence.
- **index (索引):** An integer value used to select an item in a sequence, such as a character in a string. In Python indices start from 0.
- **slice (切片):** A part of a string specified by a range of indices.
- **empty string (空字串):** A string with no characters and length 0, represented by two quotation marks.
- **immutable (不可變的):** The property of a sequence whose items cannot be changed.
- **traverse (遍歷):** To iterate through the items in a sequence, performing a similar operation on each.
- **search (搜尋):** A pattern of traversal that stops when it finds what it is looking for.
- **counter (計數器):** A variable used to count something, usually initialized to zero and then incremented.
- **invocation (調用):** A statement that calls a method.
- **optional argument (任意引數):** A function or method argument that is not required.

練習1

完成 Exercise 8.1 至 8.5 的練習

8.13 Exercises

Exercise 8.1

Read the documentation of the string methods at http://docs.python.org/3/library/stdtypes.html#string-methods.

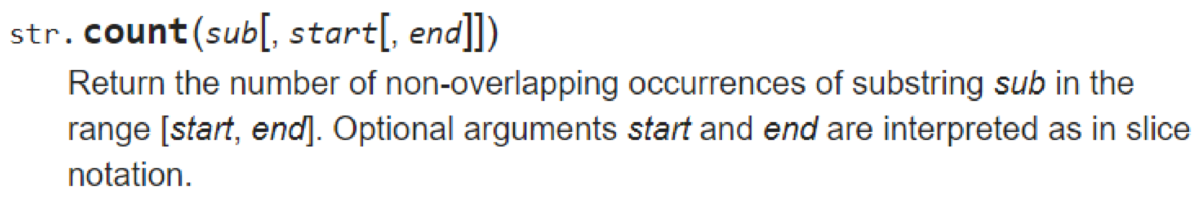
You might want to experiment with some of them to make sure you understand how they work. `strip` and `replace` are particularly useful.

The documentation uses a syntax that might be confusing. For example, in `find(sub[, start[, end]])`, the brackets indicate optional arguments. So `sub` is required, but `start` is optional, and if you include `start`, then `end` is optional.

Exercise 8.2

There is a string method called `count` that is similar to the function in Section 8.7. Read the documentation of this method and write an invocation that counts the number of `a's` in 'banana'.

https://docs.python.org/3/library/stdtypes.html#string-methods



`str.`**`count`**(*sub*[, *start*[, *end*]])

Return the number of non-overlapping occurrences of substring *sub* in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation.

Exercise 8.3

A string slice can take a third index that specifies the "step size"; that is, the number of spaces between successive characters. A step size of 2 means every other character; 3 means every third, etc.

```
>>> fruit = 'banana'
>>> fruit[0:5:2]
'bnn'
```

A step size of -1 goes through the word backwards, so the slice `[::-1]` generates a reversed string.

Use this idiom to write a one-line version of `is_palindrome` from Exercise 6.3.

Exercise 8.4

The following functions are all intended to check whether a string contains any lowercase letters, but at least some of them are wrong. For each function, describe what the function actually does (assuming that the parameter is a string).

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'

def any_lowercase3(s):
    for c in s:
        flag = c.islower()
    return flag

def any_lowercase4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag

def any_lowercase5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

Exercise 8.5

A Caesar cypher is a weak form of encryption that involves "rotating" each letter by a fixed number of places. To rotate a letter means to shift it through the alphabet, wrapping around to the beginning if necessary, so 'A' rotated by 3 is 'D' and 'Z' rotated by 1 is 'A'.

To rotate a word, rotate each letter by the same amount. For example, "cheer" rotated by 7 is "jolly" and "melon" rotated by -10 is "cubed". In the movie **2001: A Space Odyssey**, the ship computer is called HAL, which is IBM rotated by -1.

Write a function called `rotate_word` that takes a string and an integer as parameters, and returns a new string that contains the letters from the original string rotated by the given amount.

You might want to use the built-in function `ord`, which converts a character to a numeric code, and `chr`, which converts numeric codes to characters. Letters of the alphabet are encoded in alphabetical order, so for example:

```
>>> ord('c') - ord('a')
2
```

Because 'c' is the two-eth letter of the alphabet. But beware: the numeric codes for upper case letters are different.

Potentially offensive jokes on the Internet are sometimes encoded in ROT13, which is a Caesar cypher with rotation 13. If you are not easily offended, find and decode some of them. Solution: http://thinkpython2.com/code/rotate.py.

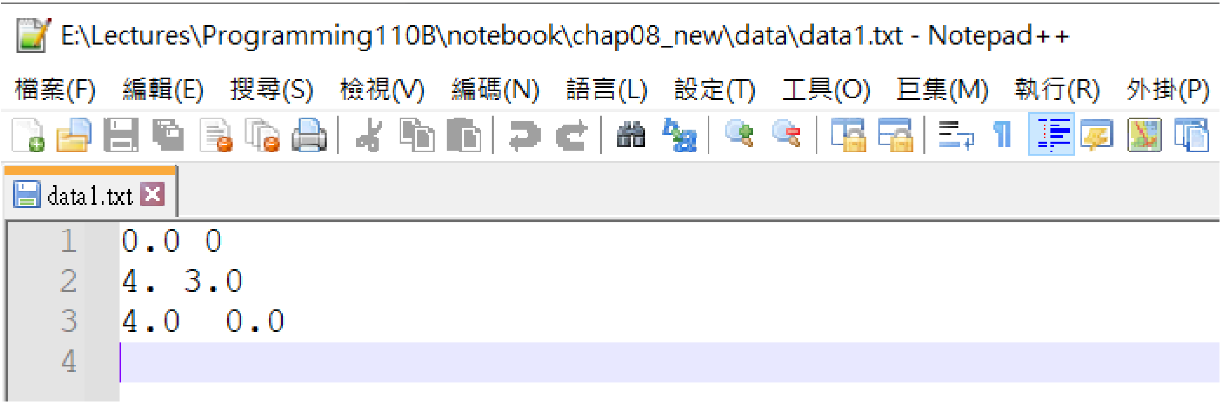
練習2

1. polyline (折線) 由連接連續 vertices (頂點) 的 segments (線段) 組成，請自行準備一個檔案紀錄折線之各頂點坐標，可參考 data1.txt 的格式，若折線的起點和終點相同且線段沒有交錯，將構成一個封閉的多邊形，例如：data2.txt。
2. 輸入的坐標點須至少5個點，且能構成一個多邊形，儲存坐標資料檔名為 data3.txt。
3. 撰寫Python程式讀取 data3.txt，計算多邊形的周長與面積。
4. 輸出資料格式如下：
```
point 1: x = 0.0, y = 0.0
point 2: x = 4.0, y = 3.0
point 3: x = 4.0, y = 0.0
point 4: x = 0.0, y = 0.0
```

```
length = 12.0
area = 6.0
```

提示：(1) n 個頂點會構成 $n-1$ 個線段，利用迴圈讀取每一個點的 (x, y) 坐標資料，需記住前一個點的 (x_p, y_p) 坐標才能計算線段的長度；(2)計算多邊形面積可以先將多邊形切割成相鄰兩點與 X 軸的梯形，然後將每一個梯形的面積相加，方法請參考 plot 筆記本。

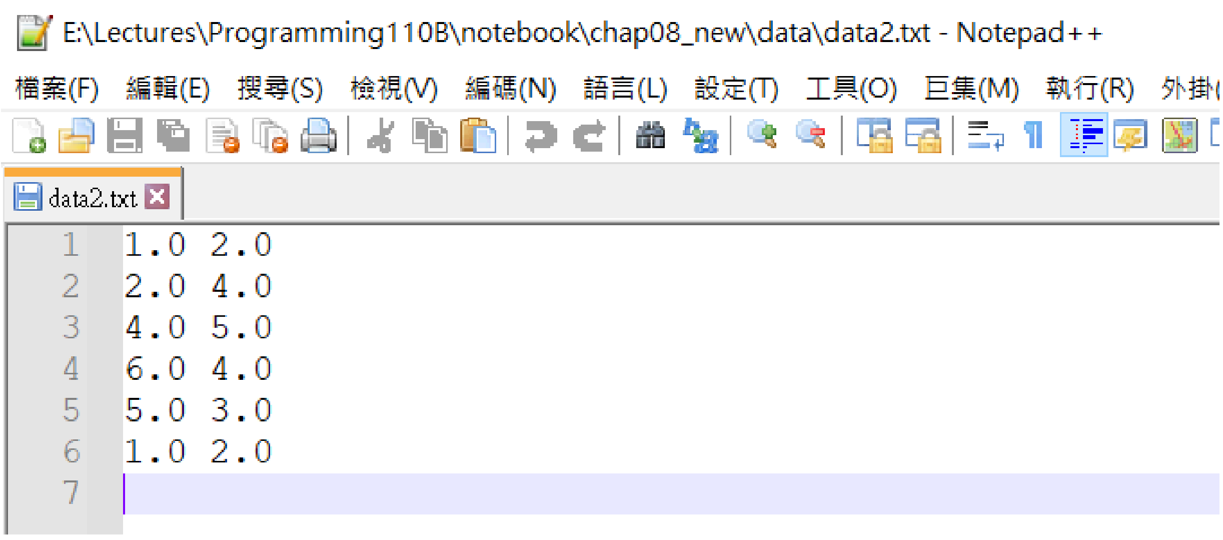
data1.txt 檔案內容：


E:\Lectures\Programming110B\notebook\chap08_new\data\data1.txt - Notepad++
檔案(F) 編輯(E) 搜尋(S) 檢視(V) 編碼(N) 語言(L) 設定(T) 工具(O) 巨集(M) 執行(R) 外掛(P)
data1.txt

```
0.0 0
4. 3.0
4.0  0.0

```

data2.txt 檔案內容：

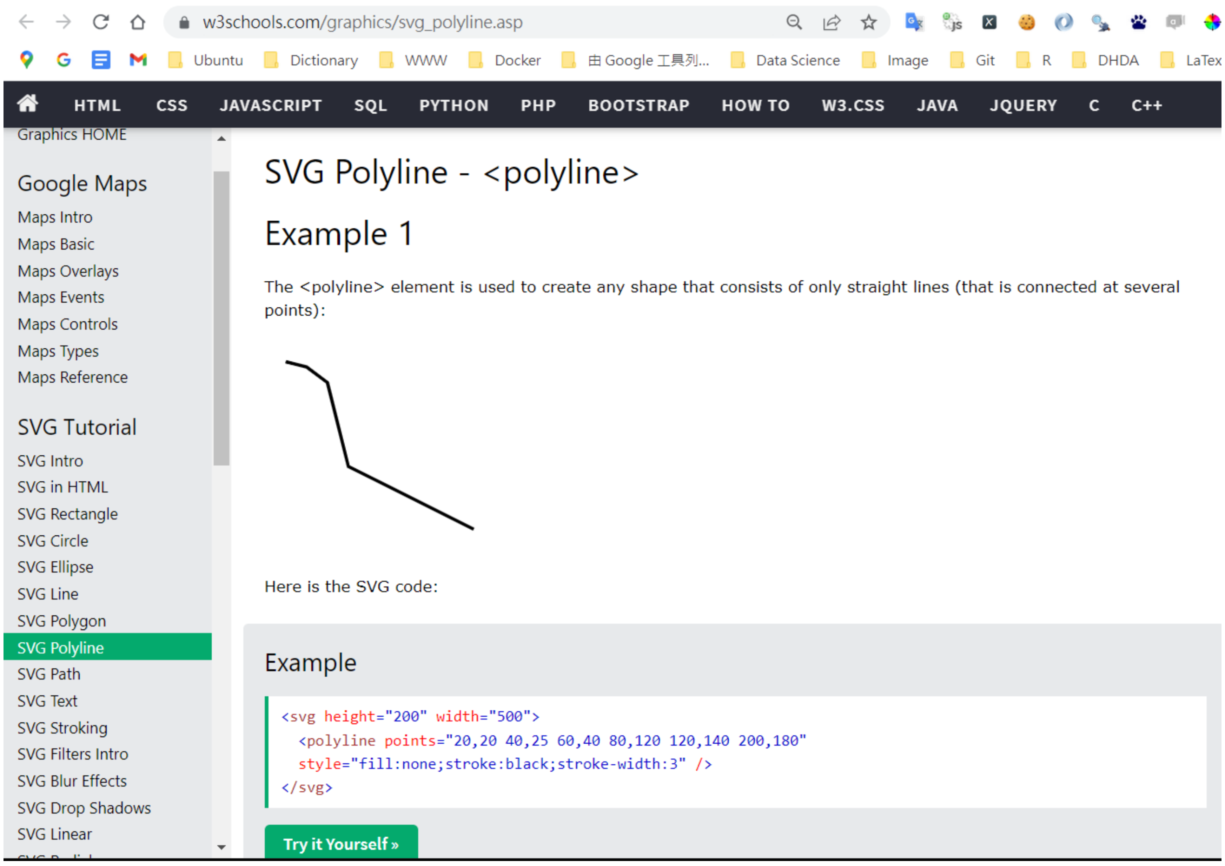

E:\Lectures\Programming110B\notebook\chap08_new\data\data2.txt - Notepad++
檔案(F) 編輯(E) 搜尋(S) 檢視(V) 編碼(N) 語言(L) 設定(T) 工具(O) 巨集(M) 執行(R) 外掛(
data2.txt

```
1.0 2.0
2.0 4.0
4.0 5.0
6.0 4.0
5.0 3.0
1.0 2.0

```

polyline 參考資料：

SVG Polyline - https://www.w3schools.com/graphics/svg_polyline.asp



SVG Polyline - <polyline>

Example 1

The <polyline> element is used to create any shape that consists of only straight lines (that is connected at several points):

Here is the SVG code:

Example

```
<svg height="200" width="500">
  <polyline points="20,20 40,25 60,40 80,120 120,140 200,180"
  style="fill:none;stroke:black;stroke-width:3" />
</svg>
```

常用的系統指令：

- **pwd**: 列印目前的工作路徑 (print working directory)，路徑 (path) 亦可稱為目錄 (directory) 或資料夾 (folder)
- **cd**: 切換路徑 (change directory)
- **ls**: 列出資料夾中的內容

參考資料：

IPython and Shell Commands https://jakevdp.github.io/PythonDataScienceHandbook/01.05-ipython-and-shell-commands.html

切換目錄至 ./data

In [25]:

```
# change directory
%cd "./data"
```

```
e:\Lectures\Programming110B\notebook\chap08_new\data
```

列印目前工作目錄名稱

In [26]:

```
pwd
```

Out[26]:

```
'e:\\Lectures\\Programming110B\\notebook\\chap08_new\\data'
```

列出目錄中的內容 (兩個不同指令)

In [27]:

```
ls *.txt
```

```
 磁碟區 E 中的磁碟是 DATA
 磁碟區序號:  30E7-0E19

 e:\Lectures\Programming110B\notebook\chap08_new\data 的目錄
```

```
2022/05/02  上午 12:13                25 data1.txt
               1 個檔案              25 位元組
               0 個目錄  1,604,855,377,920 位元組可用
```

In [28]:

```
# list the files with extension "txt"
!dir *.txt
```

```
 磁碟區 E 中的磁碟是 DATA
 磁碟區序號:  30E7-0E19

 e:\Lectures\Programming110B\notebook\chap08_new\data 的目錄

2022/05/02  上午 12:13                25 data1.txt
               1 個檔案              25 位元組
               0 個目錄  1,604,855,377,920 位元組可用
```

列印文字檔案內容

In [29]:

```
!type data1.txt
```

```
0.0 0
4. 3.0
4.0  0.0
```

以 for 迴圈讀取檔案內容

以下程式示範如何用 for 迴圈讀取 data1.txt (注意：檔案路徑名稱為 "./data/data1.txt")

參考資料：
String Methods:

https://docs.python.org/3/library/stdtypes.html#string-methods

- **str.find()**
- **str.format()**
- **str.rfind()**
- **str.split()**
- **str.strip()**

`str.`**`find`**`(`*`sub`*`[, `*`start`*`[, `*`end`*`]])`

Return the lowest index in the string where substring *sub* is found within the slice `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` if *sub* is not found.

Note: The `find()` method should be used only if you need to know the position of *sub*. To check if *sub* is a substring or not, use the `in` operator:

```
>>> 'Py' in 'Python'
True
```

`str.`**`format`**`(*`*`args`*`, **`*`kwargs`*`)`

Perform a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces `{}`. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

See Format String Syntax for a description of the various formatting options that can be specified in format strings.

Note: When formatting a number (`int`, `float`, `complex`, `decimal.Decimal` and subclasses) with the `n` type (ex: `'{:n}'.format(1234)`), the function temporarily sets the `LC_CTYPE` locale to the `LC_NUMERIC` locale to decode `decimal_point` and `thousands_sep` fields of `localeconv()` if they are non-ASCII or longer than 1 byte, and the `LC_NUMERIC` locale is different than the `LC_CTYPE` locale. This temporary change affects other threads.

Changed in version 3.7: When formatting a number with the `n` type, the function sets temporarily the `LC_CTYPE` locale to the `LC_NUMERIC` locale in some cases.

`str.`**`rfind`**`(`*`sub`*`[, `*`start`*`[, `*`end`*`]])`

Return the highest index in the string where substring *sub* is found, such that *sub* is contained within `s[start:end]`. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` on failure.

`str.`**`split`**`(`*`sep=None`*`, `*`maxsplit=- 1`*`)`

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns `['']`.

For example:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
```

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

For example:

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> '   1   2   3   '.split()
['1', '2', '3']
```

`str.`**`split`**`(`*`sep=None`*`, `*`maxsplit=- 1`*`)`

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most `maxsplit+1` elements). If *maxsplit* is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The *sep* argument may consist of multiple characters (for example, `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified separator returns `['']`.

For example:

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
```

If *sep* is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

For example:

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> '   1   2   3   '.split()
['1', '2', '3']
```

In [30]:
```python
# open a file for input
in_file = open('data1.txt')

# read from input file one line at a time
for aLine in in_file:
    print(aLine, "len(aLine) = ", len(aLine))
    # strip the trailing spaces and newline characters
    s = aLine.strip()
    print(s, "len(s) = ", len(s))
    print(aLine, "len(aLine) = ", len(aLine))

in_file.close()
```

```
0.0 0
 len(aLine) =  6
0.0 0 len(s) =  5
0.0 0
 len(aLine) =  6
4. 3.0
 len(aLine) =  7
4. 3.0 len(s) =  6
4. 3.0
 len(aLine) =  7
4.0  0.0
 len(aLine) =  9
4.0  0.0 len(s) =  8
4.0  0.0
 len(aLine) =  9
```

In [31]:
```python
# open a file for input
in_file = open('data1.txt')

# read from input file one line at a time
for aLine in in_file:
    print('[{}], length: {}'.format(aLine, len(aLine)))
    # strip the trailing spaces and newline characters
    s = aLine.strip()
    print('[{}], length: {}'.format(s, len(s)))
    print('[{}], length: {}'.format(aLine, len(aLine)))

in_file.close()
```

```
[0.0 0
], length: 6
[0.0 0], length: 5
[0.0 0
], length: 6
[4. 3.0
], length: 7
[4. 3.0], length: 6
[4. 3.0
], length: 7
[4.0  0.0
], length: 9
[4.0  0.0], length: 8
[4.0  0.0
], length: 9
```

以 **strip()** 去掉每一行最後的跳行符號，然後列印

In [32]:
```python
# open a file for input
in_file = open('data1.txt')

# read from input file one line at a time
for aLine in in_file:
    # strip the trailing spaces and newline characters
    s = aLine.strip()
    print('[{}], length: {}'.format(s, len(s)))

in_file.close()
```

```
[0.0 0], length: 5
[4. 3.0], length: 6
[4.0  0.0], length: 8
```

以 ord() 找出每一個字母的編碼 (ASCII code)

In [33]:
```
# open a file for input
in_file = open('data1.txt')

aLine = in_file.readline()     # read one line of data
print('[{}], length: {}'.format(aLine, len(aLine)))

# read from input file one line at a time
for c in aLine:
    print('[{}], code: {}'.format(c, ord(c)))

in_file.close()
```

```
[0.0 0
], length: 6
[0], code: 48
[.], code: 46
[0], code: 48
[ ], code: 32
[0], code: 48
[
], code: 10
```

以 strip() 去掉每一行最後的跳行符號，然後以 split() 分隔字串

In [16]:
```
# open a file for input
in_file = open('data1.txt')

# read from input file one line at a time
for aLine in in_file:

    # strip the trailing spaces and newline characters
    line = aLine.strip()
    s1, s2 = line.split()
    print('s1 = {}, length: {}, type = {}'.format(s1, len(s1), type(s1)))
    print('s2 = {}, length: {}, type = {}'.format(s2, len(s2), type(s2)))

in_file.close()
```

```
s1 = 0.0, length: 3, type = <class 'str'>
s2 = 0, length: 1, type = <class 'str'>
s1 = 4., length: 2, type = <class 'str'>
s2 = 3.0, length: 3, type = <class 'str'>
s1 = 4.0, length: 3, type = <class 'str'>
s2 = 0.0, length: 3, type = <class 'str'>
```

以 strip() 去掉每一行最後的跳行符號，然後以 split() 分隔字串，再轉換為 float

In [17]:
```
# open a file for input
in_file = open('data1.txt')

# read from input file one line at a time
for aLine in in_file:

    # strip the trailing spaces and newline characters
    line = aLine.strip()
    s1, s2 = line.split()
    x = float(s1)
    y = float(s2)
    print('x = {}, length: {}, type = {}'.format(x, len(s1), type(x)))
    print('y = {}, length: {}, type = {}'.format(y, len(s2), type(y)))

in_file.close()
```

```
x = 0.0, length: 3, type = <class 'float'>
y = 0.0, length: 1, type = <class 'float'>
x = 4.0, length: 2, type = <class 'float'>
y = 3.0, length: 3, type = <class 'float'>
x = 4.0, length: 3, type = <class 'float'>
y = 0.0, length: 3, type = <class 'float'>
```

切換目錄

In [35]:

```
pwd
```

Out[35]:

```
'e:\\Lectures\\Programming110B\\notebook\\chap08_new\\data'
```

In [36]:

```
%cd ..
```

```
e:\Lectures\Programming110B\notebook\chap08_new
```

切換目錄後，輸入的檔案須有完整的路徑名稱才能找到檔案

In [37]:

```
# open a file for input
file_name = './data/data1.txt'
in_file = open(file_name)

# read from input file one line at a time
for aLine in in_file:

    # strip the trailing spaces and newline characters
    line = aLine.strip()
    s1, s2 = line.split()
    x = float(s1)
    y = float(s2)
    print('x = {}, length: {}, type = {}'.format(x, len(s1), type(x)))
    print('y = {}, length: {}, type = {}'.format(y, len(s2), type(y)))

in_file.close()
```

```
x = 0.0, length: 3, type = <class 'float'>
y = 0.0, length: 1, type = <class 'float'>
x = 4.0, length: 2, type = <class 'float'>
y = 3.0, length: 3, type = <class 'float'>
x = 4.0, length: 3, type = <class 'float'>
y = 0.0, length: 3, type = <class 'float'>
```

In [ ]: