BIOS6642

Strings

Fuyong Xing

Department of Biostatistics and Informatics

Colorado School of Public Health

University of Colorado Anschutz Medical Campus

Outline

- **Strings**
- Strings as objects

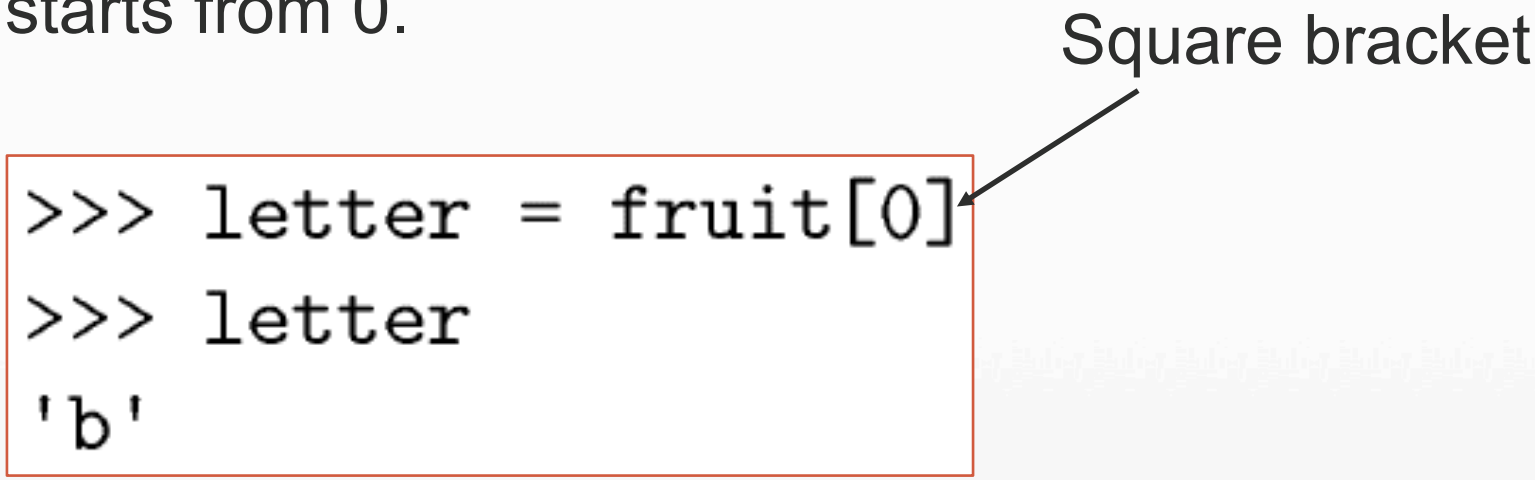
Strings

- A string is a sequence of characters.
- It is an ordered collection of characters.
- It is usually defined by using either single quotes (') or double quotes ("). Triple quotes are used to define multi-line strings.

```
>>> fruit = 'banana'
>>> type(fruit)
<class 'str'>
```

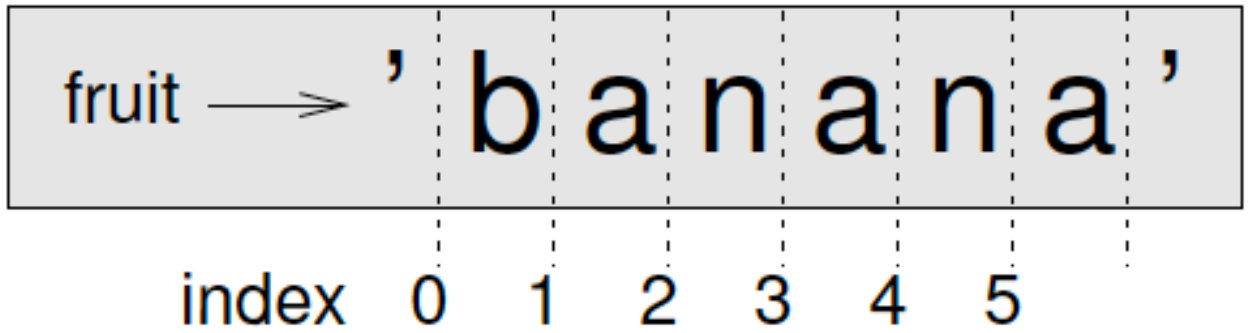
Strings

- **Indexing**: We can access a certain character in a string by using an index.
- An index is an integer value used to select an item in a sequence, such as an element in a string.
- For a string, an index represents the distance from the beginning of the string.
- The index value starts from 0.



Strings

- The index value starts from 0.



Strings

- We can also use an expression that contains variables and operators to access characters.

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

Strings

- Length of a string, i.e., the number of characters, can be achieved by using a built-in function, ***len***.

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

Strings

- Access the last character of a string.

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

- How about we write, ***fruit[length]***?

Strings

- Access the last character of a string

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

- How about we write, ***fruit[length]****?*
- We can also use a negative index to access the last character: ***fruit[-1]****.*

Strings

- **Slicing**: select a segment of a string
  - The operator *[n:m]* returns the part of the string from the “n-th” character to the “m-th” character, including the first but excluding the last.

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

Strings

- **Slicing**: select a segment of a string
  - If the first index is omitted, the slice starts at the beginning of the string.
  - If the second index is omitted, the slice goes to the end of the string.

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

Strings

- **Slicing**: select a segment of a string
  - If the first index is greater than or equal to the second, the result is an empty string, which has no characters and has length 0.

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

Strings

- **Slicing**: select a segment of a string
  - If the first index is greater than or equal to the second, the result is an empty string, which has no characters and has length 0.

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

  - How about ***fruit[:]***?

Strings

- Strings are immutable.

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

Strings

- Strings are immutable.

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

- Create a new string but has no effect on the original string

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

Strings

- String concatenation

Codes

```
string1 = 'Py'
string2 = 'thon'
print(string1 + string2)
```

Output

```
Python
```

- What does the following program print?

Codes

```
string1 = 'Py'
string2 = 'thon'
print(string1 * 3 + string2)
```

Strings

- String concatenation

Codes

```
string1 = 'Py'
string2 = 'thon'
print(string1 + string2)
```

Output

```
Python
```

- What does the following program print?

Codes

```
string1 = 'Py'
string2 = 'thon'
print(string1 * 3 + string2)
```

Output

```
PyPyPython
```

Strings

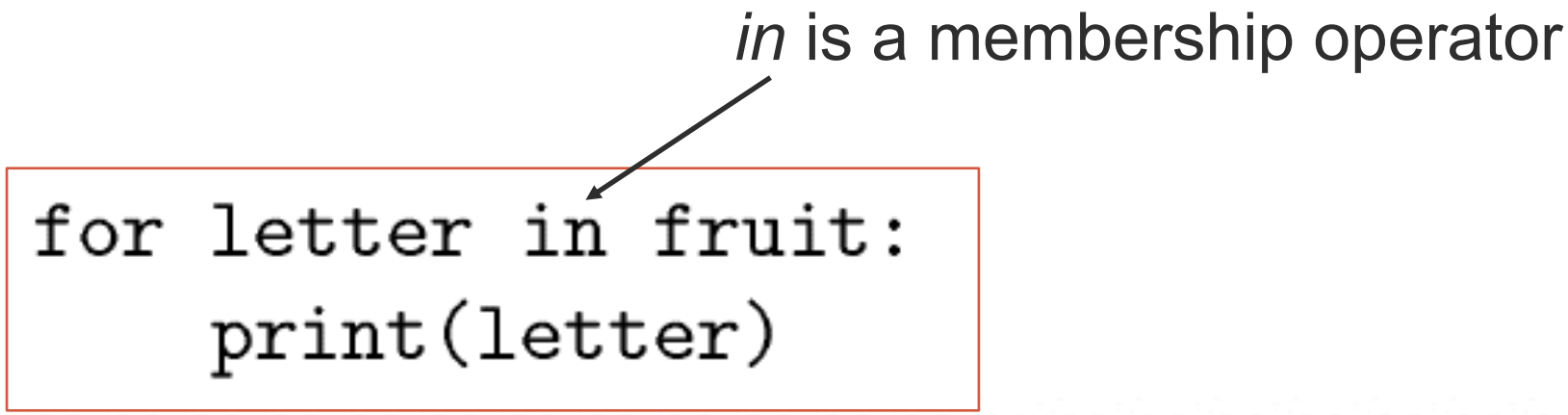
- Traversal of a string with a *while* loop

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

- This program traverses the *fruit* string and displays a character (letter) on one line.

Strings

- Traversal of a string with a *for* loop



- This program also traverses the *fruit* string and displays a character (letter) on one line.

Strings

- The *in* operator
  - ***x in s*** evaluates to *True* if ***x*** is a member of ***s***, and *False* otherwise.

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

Strings

- The *in* operator: print common characters of two strings

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

Codes

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

Output

Strings

- Search for a specific item in a string

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

- This function takes a string and a character as input, and finds the index where that character appears.

Strings

- Counting

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

- This program counts the number of times the character 'a' appears in the string word, 'banana'.

Strings

- String comparison: relational operators are applicable to strings.
  - Compare the characters one by one: if the first characters of two strings are equal, then it compares the second characters.
  - All the uppercase letters come before all the lowercase letters.

```
word = "Pineapple"

if word < "banana":
    print("Your word, " + word + ", comes before banana.")
elif word > "banana":
    print("Your word, " + word + ", comes after banana.")
else:
    print("Yes, we have no bananas!")
```

Execute this statement

Outline

- Strings
- **Strings as objects**

Strings as Objects

- Strings (, integers, and floating-point numbers) are objects in Python.
- An object is an instance of a class.
- 'banana' has a type of *str*, which is a built-in string class in Python.
- An object consists of data and methods.
  - For a string, data consist of the sequence of characters that make up the string.
  - Methods are a set of operations on the data.

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

Method call: convert letters to uppercase

Strings as Objects

- Method call

- *object* is an expression that represents an object.
- The period, pronounced dot, associates an object expression with the method to be called.
- *methodname* is the name of the method to execute.
- The *parameterlist* is comma-separated list of parameters to the method. For some methods, the parameter list may be empty, but the parentheses always are required.

Strings as Objects

- Method call

Codes

```
name = input("Please enter your name: ")
print("Hello " + name.upper() + ", how are you?")
```

Output

```
Please enter your name: Rick
Hello RICK, how are you?
```

Strings as Objects

- Method call: use the *rjust* string method to right justify a string padded with a specified character.

Codes

```
word = "ABCD"
print(word.rjust(10, "*"))
print(word.rjust(3, "*"))
print(word.rjust(15, ">"))
print(word.rjust(10))
```

Output

```
******ABCD
ABCD
>>>>>>>>>>>ABCD
      ABCD
```

Strings as Objects

- Method call from string literals and variables

```
>>> 'aBcDeFgHiJ'.upper()
'ABCDEFGHIJ'
>>> 'This is a sentence.'.rjust(25, '-')
'------This is a sentence.'
```

```
>>> '{0} {1}'.format(23, 9)
'23 9'
>>> s = '{0} {1}'
>>> s.format(23, 9)
'23 9'
```

Strings as Objects

- Some string methods

`str` Methods
`upper` Returns a copy of the original string with all the characters converted to uppercase
`lower` Returns a copy of the original string with all the characters converted to lower case
`rjust` Returns a string right justified within an area padded with a specified character which defaults to a space
`ljust` Returns a string left justified within an area padded with a specified character which defaults to a space
`center` Returns a copy of the string centered within an area of a given width and optional fill characters; fill characters default to spaces

Strings as Objects

- Some string methods

`strip`	Returns a copy of the given string with the leading and trailing whitespace removed; if provided an optional string, the `strip` function strips leading and trailing characters found in the parameter string
`startswith`	Determines if the string parameter is a prefix of the invoking string
`endswith`	Determines if the string parameter is a suffix of the invoking string
`count`	Determines the number times the string parameter is found as a substring within the invoking string; the count includes only non-overlapping occurrences
`find`	Returns the lowest index where the string parameter is found as a substring of the invoking string; returns −1 if the parameter is not a substring of the invoking string
`format`	Embeds formatted values in a string using {0}, {1}, etc. position parameters

Strings as Objects

- String method examples

Codes

```
#  Strip leading and trailing whitespace and count substrings
s = "     ABCDEFGHBCDIJKLMNOPQRSBCDTUVWXYZ      "
print("[", s, "]", sep="")
s = s.strip()
print("[", s, "]", sep="")

#  Count occurrences of the substring "BCD"
print(s.count("BCD"))
```

Output

```
[     ABCDEFGHBCDIJKLMNOPQRSBCDTUVWXYZ      ]
[ABCDEFGHBCDIJKLMNOPQRSBCDTUVWXYZ]
3
```

Strings as Objects

- Note that the method, *strip*, does not change a given string, but returns a new string instead.

```
s = "    ABC    "
s.strip()     # s is unchanged
print("<" + s + ">")    # Prints <    ABC    >. not <ABC>
s = "    ABC    "
s = s.strip()     # Note the reassignment
print("<" + s + ">")    # Prints <ABC>
```

Readings

- Think Python, chapter 8

References

- *Think Python: How to Think Like a Computer Scientist*, 2nd edition, 2015, by Allen Downey
- *Python Cookbook*, 3rd edition, by David Beazley and Brian K. Jones, 2013
- *Python for Data Analysis*, 2nd edition, by Wes McKinney, 2018
- *Fundamentals of Python Programming,* by Richard L. Halterman