Computer Programming I

Binnur Kurt, PhD

BAU Bahçeşehir University

binnur.kurt@rc.bau.edu.tr

MODULE 5

STRINGS

Strings

> Python has powerful and flexible built-in string processing capabilities

> You can write *string literals* using either single quotes ' or double quotes ":

```
a = 'one way of writing a string'
b = "another way"
```

> For multiline strings with line breaks, you can use triple quotes, either ''' or """:

```
c = """
This is a longer string that
spans multiple lines
"""
```

> The line breaks after """ and after lines are included

```
In [55]: c.count('\n')
Out[55]: 3
```

Strings

> Python strings are immutable; you cannot modify a string:

```
In [56]: a = 'this is a string'

In [57]: a[10] = 'f'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-57-5ca625d1e504> in <module>()
----> 1 a[10] = 'f'
TypeError: 'str' object does not support item assignment

In [58]: b = a.replace('string', 'longer string')

In [59]: b
Out[59]: 'this is a longer string'
```

> After this operation, the variable a is unmodified:

```
In [60]: a
Out[60]: 'this is a string'
```

Strings

> Many Python objects can be converted to a string using the **`str`** function:

```
In [61]: a = 5.6

In [62]: s = str(a)

In [63]: print(s)
5.6
```

Strings

> Strings are a sequence of Unicode characters and can be treated like other sequences, e.g. lists and tuples:

```
In [64]: s = 'python'

In [65]: list(s)
Out[65]: ['p', 'y', 't', 'h', 'o', 'n']

In [66]: s[:3]
Out[66]: 'pyt'

In [17]: x = '\u20ba'
         print(x)

         ₺
```

slicing

Strings

- The backslash character **** is an *escape character*
- It is used to specify special characters like newline \n or Unicode characters.
- To write a string literal with backslashes, you need to escape them:

```
In [67]: s = '12\\34'

In [68]: print(s)
12\34
```

Strings

- If you have a string with a lot of backslashes and no special characters, you might find this a bit annoying.
- Fortunately, you can preface the leading quote of the string with **r**
  - The r stands for *raw*
  - It means that the characters should be interpreted as is:

```
In [69]: s = r'this\has\no\special\characters'

In [70]: s
Out[70]: 'this\\has\\no\\special\\characters'
```

Strings

> Adding two strings together concatenates them and produces a new string:

```
In [71]: a = 'this is the first half '

In [72]: b = 'and this is the second half'

In [73]: a + b
Out[73]: 'this is the first half and this is the second half'
```

Strings

> String templating or formatting is important
> String objects have a format method
  – Used to substitute formatted arguments into the string, producing a new string:

```
In [74]: template = '{0:.2f} {1:s} are worth US${2:d}'
```

  – **{0:.2f}** means to format the first argument as a floating-point number with two decimal places.
  – **{1:s}** means to format the second argument as a string.
  – **{2:d}** means to format the third argument as an exact integer.

Strings

> To substitute arguments for these format parameters, pass a sequence of arguments to the **`format`** method:

```
In [75]: template.format(4.5560, 'Argentine Pesos', 1)
Out[75]: '4.56 Argentine Pesos are worth US$1'
```

> String formatting is a deep topic
  - there are multiple methods and numerous options
  - tweaks available to control how values are formatted in the resulting string
  - To learn more, consult the official Python documentation: https://docs.python.org/3/

Bytes and Unicode

> In modern Python (Python 3.0+), Unicode has become the first-class string type to enable more consistent handling of ASCII and non-ASCII text.

```
In [76]: val = "español"

In [77]: val
Out[77]: 'español'
```

> You can convert this Unicode string to its UTF-8 bytes representation using the **`encode`** method

```
In [78]: val_utf8 = val.encode('utf-8')

In [79]: val_utf8
Out[79]: b'espa\xc3\xb1ol'

In [80]: type(val_utf8)
Out[80]: bytes
```

Bytes and Unicode

> Assuming you know the Unicode encoding of a **`bytes`** object, you can go back using the **`decode`** method:

```
In [81]: val_utf8.decode('utf-8')
Out[81]: 'español'
```

> While it's become preferred to use UTF-8 for any encoding, for historical reasons you may encounter data in any number of different encodings:

```
In [82]: val.encode('latin1')
Out[82]: b'espa\xf1ol'

In [83]: val.encode('utf-16')
Out[83]: b'\xff\xfee\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'

In [84]: val.encode('utf-16le')
Out[84]: b'e\x00s\x00p\x00a\x00\xf1\x00o\x00l\x00'
```

Bytes and Unicode

> It is most common to encounter bytes objects in the context of working with files, where implicitly decoding all data to Unicode strings may not be desired.

> Though you may seldom need to do so, you can define your own byte literals by prefixing a string with b:

```
In [85]: bytes_val = b'this is bytes'

In [86]: bytes_val
Out[86]: b'this is bytes'

In [87]: decoded = bytes_val.decode('utf8')

In [88]: decoded  # this is str (Unicode) now
Out[88]: 'this is bytes'
```

Comparison Operators for Strings

- Strings may be compared with the comparison operators.
- Recall that strings are compared based on their underlying integer numeric values.
- So uppercase letters compare as less than lowercase letters because uppercase letters have lower integer values.

```
print(f'A: {ord("A")}; a: {ord("a")}')
```

```
A: 65; a: 97
```

Comparison Operators for Strings

- Let's compare the strings **`'Orange'`** and **`'orange'`** using the comparison operators:

```
'Orange' == 'orange'
```

```
False
```

```
'Orange' != 'orange'
```

```
True
```

```
'Orange' < 'orange'
```

```
True
```

```
'Orange' <= 'orange'
```

```
True
```

```
'Orange' > 'orange'
```

```
False
```

```
'Orange' >= 'orange'
```

```
False
```

Searching for Substrings

> String method count returns the number of times its argument occurs in the string on which the method is called

```
sentence = 'to be or not to be that is the question'
```

```
sentence.count('to')
```

```
2
```

> If you specify as the second argument a ***start_index***, **`count`** searches only the slice **`string[start_index:]`**—that is, from ***start_index*** through end of the string:

```
sentence.count('to', 12)
```

```
1
```

Searching for Substrings

> If you specify as the second and third arguments the ***start_index*** and ***end_index***, **`count`** searches only the slice **`string[start_index:end_index]`**—that is, from ***start_index*** up to, but not including, ***end_index***:

```
sentence.count('that', 12, 25)
```

```
1
```

Locating a Substring in a String

> String method **index** searches for a substring within a string and returns the first index at which the substring is found; otherwise, a **ValueError** occurs

```
sentence.index('be')
```

```
3
```

```
sentence.rindex('be')
```

```
16
```

> String methods find and rfind perform the same tasks as **index** and **rindex** but, if the substring is not found, return -1 rather than causing a **ValueError**.

Determining Whether a String Contains a Substring

> If you need to know only whether a string contains a substring, use operator **in** or **not in**

```
sentence = 'to be or not to be that is the question'
```

```
'that' in sentence
```

```
True
```

```
'THAT' in sentence
```

```
False
```

```
'THAT' not in sentence
```

```
True
```

Locating a Substring at the Beginning or End of a String

> String methods **`startswith`** and **`endswith`** return **`True`** if the string starts with or ends with a specified substring:

```
sentence = 'to be or not to be that is the question'
```

```
sentence.startswith('to')
```

```
True
```

```
sentence.startswith('be')
```

```
False
```

```
sentence.endswith('question')
```

```
True
```

```
sentence.endswith('quest')
```

```
False
```

Replacing Substrings

> A common text manipulation is to locate a substring and replace its value.
> Method **`replace`** takes two substrings.
> It searches a string for the substring in its first argument and replaces each occurrence with the substring in its second argument.
> The method returns a new string containing the results.

```
values = '1\t2\t3\t4\t5'
```

```
values.replace('\t', ',')
```

```
'1,2,3,4,5'
```

Removing Leading and Trailing Whitespace

```
sentence = '\t \n This is a test string. \t\t \n'
```

```
sentence.strip()
```

```
'This is a test string.'
```

```
sentence.lstrip()
```

```
'This is a test string. \t\t \n'
```

```
sentence.rstrip()
```

```
'\t \n This is a test string.'
```

Changing Character Case

```
'happy birthday'.capitalize()
```

```
'Happy birthday'
```

```
'artificial intelligence in algorithmic trading'.title()
```

```
'Artificial Intelligence In Algorithmic Trading'
```

Splitting Strings

```
letters = 'A, B, C, D'
```

```
letters.split(',')
```

```
['A', ' B', ' C', ' D']
```

```
list(map(lambda s: s.strip(), letters.split(',')))
```

```
['A', 'B', 'C', 'D']
```

```
letters.split(', ')
```

```
['A', 'B', 'C', 'D']
```

Joining Strings

```
letters_list = ['A', 'B', 'C', 'D']
```

```
','.join(letters_list)
```

```
'A,B,C,D'
```

```
','.join([str(i) for i in range(10)])
```

```
'0,1,2,3,4,5,6,7,8,9'
```

partition

- String method **partition** splits a string into a tuple of three strings based on the method's separator argument.
  - the part of the original string before the separator,
  - the separator itself,
  - the part of the string after the separator.

partition

```
tokens= 'bitcoin: 7800, 7900, 8100'.partition(': ')
```

```
print(tokens)
```

```
('bitcoin', ': ', '7800, 7900, 8100')
```

```
bitcoin_prices = tokens[2].split(', ')
```

```
print(bitcoin_prices)
```

```
['7800', '7900', '8100']
```

splitlines

> Method **splitlines** returns a list of new strings representing the lines of text split at each newline character in the original string.

```
lines = """This is line 1
This is line2
This is line3
This is line4
This is line5"""
```

```
lines.splitlines()
```

```
['This is line 1',
 'This is line2',
 'This is line3',
 'This is line4',
 'This is line5']
```

Characters and Character-Testing Methods

```
'-27'.isdigit()
```

```
False
```

```
'27'.isdigit()
```

```
True
```

```
'A9876'.isalnum()
```

```
True
```

```
'123 Main Street'.isalnum()
```

```
False
```

Characters and Character-Testing Methods

String Method	Description
`isalnum()`	Returns True if the string contains only alphanumeric characters (i.e., digits and letters).
`isalpha()`	Returns True if the string contains only alphabetic characters (i.e., letters).
`isdecimal()`	Returns True if the string contains only decimal integer characters (that is, base 10 integers) and does not contain a + or - sign.
`isdigit()`	Returns True if the string contains only digits (e.g., '0', '1', '2').
`isidentifier()`	Returns True if the string represents a valid identifier.
`islower()`	Returns True if all alphabetic characters in the string are lowercase characters (e.g., 'a', 'b', 'c').
`isnumeric()`	Returns True if the characters in the string represent a numeric value without a + or - sign and without a decimal point.
`isspace()`	Returns True if the string contains only whitespace characters.

Characters and Character-Testing Methods

String Method	Description
`istitle()`	Returns True if the first character of each word in the string is the only uppercase character in the word.
`isupper()`	Returns True if all alphabetic characters in the string are uppercase characters (e.g., 'A', 'B', 'C').