Python Programming Language – strings

Assoc. Prof. Krzysztof Małecki

kmalecki@zut.edu.pl
p. 308 (WI2)

Strings

- if a certain string of chars has to be considered as a string in Python, it must be enclosed in **single** or **double quotes**

- this also applies to individual chars

- string can be assignment for a variable just like any other data

```
firstName = 'Mike'

lastName = "Smith"
```

Lists and strings

- in Python strings can also be list items
- moreover, some list items can be strings and others can be numbers

```
team1 = ['Liam', 'Alicja']

team2 = ["Mike", "Sarah"]

mixList = [ 1, 'text' ]
```

print() and strings

```
x = 'Sue has a cat'
print(x)
y = "Mike has a dog"
print(y)
```

```
Sue has a cat
Mike has a dog
```

len() works correctly

```
x = 'Sue has a cat'
print(len(x))
x = ''
print(len(x))
```

```
13
0
```

Single and double quotes

```
y = "I'm going to McDonald's"
print(y)
z = 'I like "Python Programming Language"'
print(z)
```

```
I'm going to McDonald's
I like "Python Programming Language"
```

\ (backslash)

- it is used inside strings to "quote", i.e. precede certain characters to treat them in **a special way**

- this allows you to insert double quotes and apostrophes where this would not normally be possible

\ (backslash)

```
y = 'I\'m going to McDonald\'s'
print(y)
z = 'I like Monty Python\'s flying circus'
print(z)
x = "I like \"Monty Python's flying circus\"”
print(x)
```

```
I'm going to McDonald's
I like Monty Python's flying circus
I like "Monty Python's flying circus"
```

without \ (backslash) it will be a problem

```
z = 'I like Monty Python's flying circus'
print(z)
```

```
SyntaxError: invalid syntax
```

Remember

```
y = '\''
print(len(y))
```

Nice.... right?!

```
1
```

A multiline string – variant #1

```
z = """Line1,
line2,
line3,
line4."""
print(z)
```

```
Line1,
line2,
line3,
line4.
```

A multiline string – variant #2

```
z = '''Line1,
line2,
line3,
line4.'''
print(z)
```

```
Line1,
line2,
line3,
line4.
```

\ (backslash)

- it can also be used to write ASCII codes that do not have a readable character representation
- these include the so-called "white spaces", but not only ...

`\t`

ASCII code: 9 (0x09)

name: HT → Horizontal Tabulation

`\b`

ASCII code: 8 (0x08)

name: BS → BackSpace

and others…

\t

```
x = 'Sue\thas\ta\tcat'
print(x)
```

```
Sue	has	a	cat
```

Remember

- due to the very specific role of the \ character in the strings, in order to put it in the text itself, **it should be quoted** (i.e. doubled)

```
x = "This char: \\ is a backslash"
print(x)
```

```
This char: \ is a backslash
```

String operators

- we use (as in the case of numbers) operators to manipulate strings
- as it turns out in a moment, it may be that one and the same operator performs a different operation on numbers and on strings
- this situation is called **operator overloading**

A concatenation operator: + (plus)

- it combines the left and right arguments into one new string, for example:

```
'py' + 'thon' → 'python'

'thon' + 'py' → 'thonpy'
```

A duplicate operator: * (asterisk)

- it duplicates one argument as many times as specified in the second argument, resulting in a new string, for example:

```
'py' *2 → 'pypy'

10 * '*' → '**********'
```

`ord(x)` function

this function converts a character to the corresponding code

```
ord(x)
```

the argument: a certain sign

the result: its code

the effect: none

e.g.:

```
print(ord('A'))  →  65
```

`chr(x)` function

this function converts code to the corresponding character

```
chr(x)
```

the argument: code of a certain character

the result: this sign

the effect: none

e.g.:

```
print(chr(65))    →    A
```

Immutable data

- some data available in Python are treated as "immutable" (immutable)
- this means that their content may **not be changed**
- **strings in Python are immutable**
- however, you can freely create new values based on the values already available

Immutable data

the fact that strings are immutable in Python has several consequences:

1. the `del` instruction **does not work** on characters in strings - **its use is not allowed in this context**! However, it can be performed for the entire string
2. strings **do not have** an `append()` method
3. strings **do not have** an `insert()` method

…but everything we know about lists clipping also applies to strings

```
string = "abdefg"
print(string[1])
print(string[-1])
print(string[1:3])
print(string[3:])
print(string[:3])
```

```
b
g
bd
efg
abd
```

we will use this opportunity to show you strings clipping with a step

```
string = "abcdefghijklmnopqrstuvwxyz"
print(string[0:10:1])
print(string[::1])
print(string[1:18:2])
print(string[:-1:3])
print(string[1:-1:4])
```

```
abcdefghij
abcdefghijklmnopqrstuvwxyz
bdfhjlnpr
adgjmpsvy
bfjnrv
```

in and not in operators

```
string = "abcdefghijklmnopqrstuvwxyz"
print('f' in string)
print('F' in string)
print('1' in string)
print('f' not in string)
print('F' not in string)
print('1' not in string)
```

```
True
False
False
False
True
True
```

The left argument of `in` and `not in` operators can also be strings

```
string = "abcdefghijklmnopqrstuvwxyz"
print('ghi' not in string)
print('ghi' in string)
print('Xyz' not in string)
print('xyz' in string)
```

```
False
True
True
True
```

`min(seq)` function

gives the smallest element of the sequence

```
min(seq)
```

the argument: a certain sequence

the result: the smallest element of the sequence

e.g.:

```
print(min('aAbByYzZ'))
```
→ A

According to the ASCII table,
uppercase letters lie
in front of lowercase letters

Example

```
t = 'Sue has a cat'
print('[' + min(t) + ']')
t = "abzZ"
print(min(t))
t = [0,1,2]
print(min(t))
```

```
[ ]
Z
0
```

`max(seq)` function

gives the largest element of the sequence

```
max(seq)
```

the argument: a certain sequence

the result: the largest element of the sequence

e.g.:

```
print(max('aAbByYzZ'))
```
→ z

Example

```
t = 'Sue has a cat'
print(max(t))
t = "abzZ"
print(max(t))
t = [0,1,2]
print(max(t))
```

```
u
z
2
```

`index(element)` method

gives the position of the element in the sequence

```
seq.index(element):
```

the argument: a certain element

the result: the position of the element

e.g.:

```
print('aAbByYzZ'.index('b')) → 2
```

Example

```
t = 'Sue has a cat'
print(t.index('t'))
print(t.index('S'))
print(t.index('a'))
```

```
12
0
5
```

Attention

```
t = 'Sue has a cat'
i = t.index('A')
```

```
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    i = t.index('A')
ValueError: substring not found
```

Some more examples

```
t = [0,0,1,1,2,2]
print(t.index(0))
print(t.index(1))
print(t.index(2))
```

```
0
2
4
```

`count(element)` method

counts the occurrence of an element in the sequence

```
seq.count(element):
```

the argument: a certain element

the result: the number of occurrences of the element in the sequence

e.g.:

```
print('abcabc'.count('b')) → 2
```

Example

```
t = 'Sue has a cat'
print(t.count('t'))
print(t.count('a'))
print(t.count('A'))
```

```
1
3
0
```

Some string methods and functions (strictly!)

`capitalize()` method

returns a copy of the string with the first character changed to uppercase (if possible) and subsequent characters to lowercase (if possible)

`str.capitalize()`:

the argument: none

the result: the copy of `str` with uppercase of the first char

e.g.:

`print('sue'.capitalize())` → Sue

Example

```
print('Sue'.capitalize())
print('SUE'.capitalize())
print(' Sue'.capitalize())
print('123'.capitalize())
```

```
Sue
Sue
 sue
123
```

`center(width)` method

returns a copy of the string centered in the `width` field

`str.center(width)`:

the argument: the width of target field

the result: a copy of `str` fitted to a field of the given width

e.g.:

```
print('[' + 'sue'.center(5) + ']') → [ sue ]
```

`center(width,filler_sign)` method

returns a copy of the string centered in the `width` field

`str.center(width,filler_sign)`:

arguments: `width`: the width of target field

`filler_sign`: filler, exactly one character

the result: a copy of `str` fitted to a field of the given `width` with using the character specified in the `filler_sign`

e.g.:

`print('sue'.center(5,'*'))` → `*sue*`

Example

```
print('[' + 'Sue'.center(2) + ']')
print('[' + 'Sue'.center(3) + ']')
print('[' + 'Sue'.center(4) + ']')
print('[' + 'Sue'.center(7) + ']')
print('[' + 'Sue'.center(7, '*') + ']')
```

```
[Sue]
[Sue]
[Sue ]
[  Sue  ]
[**Sue**]
```

`endswith(suffix)` method

checks if the string ends with the string `suffix`

```
str.endswith(suffix):
```

the argument: the part of string

the result: `True / False`

e.g.:

```
print('Sue has a cat'.endswith('at')) → True
```

`find(substring)` method

looking for a `substring` in the `str`

```
str.find(substring):
```

the argument: the `substring` looked for

the result: the position of a `substring` in the string, or -1

e.g.:

```
print('Sue has a cat'.find('has')) → 4
print('Sue has a cat'.find('sue')) → -1
```

Remember

- do not use `find()` if you just want to check that a certain character is present in a string - the `in` operator **is faster!**

`find(substring,start)` method

looking for a `substring` in the `str` from the `start` point

```
str.find(substring,start):
```

the arguments: the `substring` looked for

`start` - the position from which the `substring` is looking for

the result: the position of a `substring` in the string, or -1

e.g.:

```
print('Sue has a cat'.find('a',4)) → 5
print('Sue has a cat'.find('Sue',3)) → -1
```

`find(substring,start,stop)` method

looking for a `substring` in the `str` from the `start` point but no further than `stop` point

```
str.find(substring,start,stop):
```

the arguments: the `substring` looked for
`start` - the position from which the `substring` is looking for
`stop` - the position to which the `substring` is looking for

the result: the position of a `substring` in the string, or -1

e.g.:

```
print('Sue has a cat'.find('a',4,5)) → -1
```

`isalnum()` **method** - *is alphanumeric*

checks if the string contains only letters and numbers

```
str.isalnum()
```

the argument: none

the result: `True / False`

e.g.:

```
print('Sue has a cat'.isalnum()) → False
print('Sue'.isalnum()) → True
```

`isalpha()` **method** - *is alphabetic*

checks if the string contains only letters

```
str.isalpha()
```

the argument: none

the result: `True / False`

e.g.:

```
print('Suehasacat'.isalpha()) → True
```

`isdigit()` method

checks if the string contains only decimal numbers

```
str.isdigit()
```

the argument: none

the result: `True / False`

e.g.:

```
print('20160101'.isdigit()) → True
```

`islower()` method

checks if the string contains only lowercase letters

```
str.islower()
```

the argument: none

the result: `True / False`

e.g.:

```
print('Sue'.islower()) → False
print('has'.islower()) → True
```

`isspace()` method

checks if the string contains only so-called “white signs”

```
str.isspace()
```

the argument: none

the result: `True / False`

e.g.:

```
print(' \t\n'.isspace()) → True
```

`join(list)` method

concatenates list elements into a single string, separating them with the string `str`; **list items must be strings!**

```
str.join(list)
```

the argument: `list` – a certain list

the result: **a string** made up of list elements separated by `str`

e.g.:

```
print(','.join(['1','2','3'])) → '1,2,3'
```

`list(str)` function

creates a list that contains all the characters in the string one by one

```
list(str)
```

the argument: `str` – a certain string

the result: **a list** with the length `len(str)` contains signs taken from `str`

e.g.:

```
print(list('abc')) → ['a','b','c']
```

`ljust(width)` **method** – *left justify*

returns a copy of the `str` left justified to the width

```
str.ljust(width)
```

the argument: `width` – target string width

the result: a copy of left justified string

e.g.:

```
print('['+'Sue has a cat'.ljust(20)+']') → [Sue has a cat       ]
print('['+'Sue has a cat'.ljust(20,'*')+']') → [Sue has a cat*******]
```

`lower()` method

returns a copy of the string with all letters changed to lowercase

```
str.lower()
```

the argument: none

the result: a copy of the string with the letters changed to lowercase

e.g.:

```
print('CAT'.lower()) → cat
```

`lstrip()` method – *left strip*

returns a copy of a string with leading whitespace removed

```
str.lstrip()
```

the argument: none

the result: a copy of the string with the leading white signs removed

e.g.:

```
print('[' + '   x '.lstrip() + ']')
```
→ `[x ]`

`lstrip(set)` method – *left strip*

returns a copy of a string with leading signs included in `set` removed

```
str.lstrip(set)
```

the argument: `set` – a set of signs which should be removed

the result: a copy of the string with the leading signs removed

e.g.:

```
print('www.zut.pl'.lstrip('w.'))
```
→ `zut.pl`

`replace(old,new)` method

returns a copy of the string with the substrings `old` replaced with new

```
str.replace(old,new)
```

the arguments: `old` – a substring which should be changed by new

the result: a copy of the string with some substrings replaced

e.g.:

```
print('this is this'.replace('this','it')) → it is it
print('this is this'.replace('this','it',1)) → it is this
```

maximum number of replacements

`rfind(substring)` method - *right find*

looks for a substring in a string (like `find()`, but looks backwards!)

```
str.rfind(substring)
```

the arguments: `substring` – a searched string

the result: the position of a `substring` in the string, or -1

e.g.:

```
print('this is this'.rfind('this')) → 8
print('this is there'.rfind('this',3)) → -1
print('it it it'.rfind('it',3,5)) → 3
```

the position to which `rfind()` is looking for

the position to which `rfind()` is looking for

the position from which `rfind()` is looking for

`rjust(width)` method – *right justify*

returns a copy of the `str` right justified to the width

`str.rjust(width)`

the argument: `width` – target string width

the result: a copy of right justified string

e.g.:

```
print('['+'Sue has a cat'.rjust(20)+']')        → [       Sue has a cat]
print('['+'Sue has a cat'.rjust(20,'*')+']')    → [*******Sue has a cat]
```

`rstrip()` method – *right strip*

returns a copy of the string with trailing whitespace removed

```
str.rstrip()
```

the argument: none

the result: a copy of the string with trailing white signs removed

e.g.:

```
print('[' + '   x '.rstrip() + ']') → [   x]
print('www.zut.pl'.rstrip('l.p')) → www.zut
```

`split()` method

returns a list of substrings produced by splitting the given string at the positions of white space

```
str.split()
```

the argument: none

the result: **a list** of substrings

e.g.:

```
print('Sue has a cat'.split()) → ['Sue','has','a','cat']
```

`startswith(prefix)` method

checks if the string starts with the string `prefix`

`str.startswith(prefix)`:

the argument: the part of string

the result: `True / False`

e.g.:

```
print('Sue has a cat'.startswith('at')) → False
```

`strip()` method

returns a copy of the string with leading and trailing whitespace removed

```
str.strip()
```

the argument: none

the result: a copy of the string without white space at the beginning and at the end of the `str`

e.g.:

```
print('['+' x '.strip()+']') → [x]
```

`swapcase()` method

returns a copy of a string with all lowercase letters converted to uppercase and vice versa

```
str.swapcase()
```

the argument: none

the result: a copy of the string: lowercase <=> uppercase

e.g.:

```
print('Sue'.swapcase()) → sUE
```

`title()` method

returns a copy of the string with the first letters of words changed to uppercase and the remaining letters changed to lowercase

```
str.title()
```

the argument: none

the result: a copy of the string after changes

e.g.:

```
print('the TITLE of an aRTICLE'.title())
    → The Title Of An Article
```

`upper()` method

returns a copy of the string with all letters changed to uppercase

```
str.upper()
```

the argument: none

the result: a copy of the string with the letters changed to uppercase

e.g.:

```
print('cat'.upper()) → CAT
```

strings can be compared

- strings can be compared with each other almost as well as numbers, using the same operators:

  == != > >= < <=

- comparisons of the greater – less type assume that ASCII / UNICODE character codes are used to establish the relationship, which may give surprising results

Examples

```
'Sue' == 'Sue'          → True
'Sue' == 'sue'          → False
'Ola' > 'Ala'           → True
'Example' < 'Examples'→ True
'10' == '010'           → False
'10' > '010'            → True  (???)
'10' > '8'              → False (???)
'20' < '8'              → True  (???)
'20' < '80'             → True
```

Remember

- while making comparisons, be aware of **the type of data** being compared
- in particular, do not try to compare strings with numbers - it could be a disaster!
- the only acceptable string-to-number comparisons are `==` and `!=`
- the first is always `False`, the second is always `True` ☺

String-to-number comparing

```
'10' == 10        → False
'10' != 10        → True
'10' == 1         → False
'10' != 1         → True
'10' > 10         → Error!!

Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    '10' > 10
TypeError: '>' not supported between instances of 'str' and 'int'
```

`sorted()` function

- a sequence (e.g. list, tuple) can be an argument of `sorted()`
- `sorted()` returns a sorted copy of the original sequence
- **the sequence itself remains unchanged**

Example

```
seq1 = [5,4,3,2,1]
seq2 = sorted(seq1)
print(seq1)
print(seq2)
```

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

`sort()` method

- each sequence has a `sort()` method
- the `sort()` method sorts the original sequence
- `sort()` **does not return a result**

Example

```
seq1 = [5,4,3,2,1]
print(seq1)
seq1.sort()
print(seq1)
```

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

Remember

- sorting will be done depending on the type of data being sorted
- numbers will be sorted differently from strings, even very similar looking

Example

```
print(sorted([11, 1, 8, 9, 90, 15]))

print(sorted(['11', '1', '8', '9', '90', '15']))
```

```
[1, 8, 9, 11, 15, 90]

['1', '11', '15', '8', '9', '90']
```

number → string conversion

- a number can be converted to a string representing it using the `str()` function

- such a transformation is **always** possible

```
str(0)      → '0'
str(-1)     → '-1'
str(3.14)   → '3.14'
```

string → number conversion

- a string can be converted to the corresponding number using already known functions

  `int()` and `float()`

- ... but ... **attention** - such a transformation is **not always** possible

```
int('2') → 2
int('Sue')
Traceback (most recent call last):
      File "<pyshell#0>", line 1, in <module>
      int('Sue’)
ValueError: invalid literal for int() with base 10: 'Sue'
```

Thank you for your attention

see you at the next lecture