The nature of strings in Python

Strings - a brief review

Let's do a brief review of the nature of Python's strings.

First of all, Python's strings (or simply strings, as we're not going to discuss any other language's strings) are **immutable sequences**.

It's very important to note this, because it means that you should expect some familiar behavior from them.

For example, the `len()` function used for strings returns a number of characters contained by the arguments.

Take a look at **Example 1** below.

```
1  # Example 1
2
3  word = 'by'
4  print(len(word))
5
6
7  # Example 2
8
9  empty = ''
10 print(len(empty))
11
12
13 # Example 3
14
15 i_am = 'I\'m'
16 print(len(i_am))
```

The snippet outputs `2`.

Any string can be empty. Its length is `0` then - just like in **Example 2**.

Don't forget that a backslash (`\`) used as an escape character is not included in the string's total length.

The code in **Example 3**, therefore, outputs `3`.

Run the three example codes and check.

Multiline strings

Now is a very good moment to show you another way of specifying strings inside the Python source code. Note that the syntax you already know won't let you use a string occupying more than one line of text.

For this reason, the code here is erroneous:

```
multiLine = 'Line #1
Line #2'

print(len(multiLine))
```

Fortunately, for these kinds of strings, Python offers separate, convenient, and simple syntax.

Look at this code.

```
multiLine = '''Line #1
Line #2'''

print(len(multiLine))
```

This is what it looks like.

As you can see, the string starts with **three apostrophes**, not one. The same tripled apostrophe is used to terminate it.

The number of text lines put inside such a string is arbitrary.

The snippet outputs `15`.

Count the characters carefully. Is this result correct or not? It looks okay at first glance, but when you count the characters, it doesn't.

`Line #1` contains seven characters. Two such lines comprise 14 characters. Did we lose a character? Where? How?

No, we didn't.

The missing character is simply invisible - it's a whitespace. It's located between the two text lines.

It's denoted as: `\n`.

Do you remember? It's a special (control) character used to **force a line feed** (hence its name: LF). You can't see it, but it counts.

The multiline strings can be delimited by **triple quotes**, too, just like here:

```
multiLine = """Line #1
Line #2"""

print(len(multiLine))
```

Choose the method that is more comfortable for you. Both work the same.

Operations on strings

Like other kinds of data, strings have their own set of permissible operations, although they're rather limited compared to numbers.

In general, strings can be:

- **concatenated** (joined)
- **replicated**.

```
str1 = 'a'
str2 = 'b'

print(str1 + str2)
print(str2 + str1)
print(5 * 'a')
print('b' * 4)
```

The first operation is performed by the `+` operator (note: it's not an addition) while the second by the `*` operator (note again: it's not a multiplication).

The ability to use the same operator against completely different kinds of data (like numbers vs. strings) is called **overloading** (as such an operator is overloaded with different duties).

Analyze the example:

- The `+` operator used against two or more strings produces a new string containing all the characters from its arguments (note: the order matters - this overloaded `+`, in contrast to its numerical version, is **not commutative**)
- the `*` operator needs a string and a number as arguments; in this case, the order doesn't matter - you can put the number before the string, or vice versa, the result will be the same - a new string created by the nth replication of the argument's string.

The snippet produces the following output:

```
ab
ba
aaaaa
bbbb
```

Note: shortcut variants of the above operators are also applicable for strings (`+=` and `*=`).

Operations on strings: `ord()`

If you want to know a specific character's ASCII/UNICODE code point value, you can use a function named `ord()` (as in *ordinal*).

The function needs a >strong>one-character string as its argument - breaching this requirement causes a `TypeError` exception, and returns a number representing the argument's code point.

Look at the code below and run it.

```
# Demonstrating the ord() function

ch1 = 'a'
ch2 = ' ' # space

print(ord(ch1))
print(ord(ch2))
```

The snippet outputs:

```
97
32
```

Now assign different values to `ch1` and `ch2`, e.g., `α` (Greek alpha), and `ę` (a letter in the Polish alphabet); then run the code and see what result it outputs. Carry out your own experiments.

Operations on strings: `chr()`

If you know the code point (number) and want to get the corresponding character, you can use a function named `chr()`.

The function **takes a code point and returns its character**.

Invoking it with an invalid argument (e.g., a negative or invalid code point) causes `ValueError` or `TypeError` exceptions.

Run the code below.

```
# Demonstrating the chr() function

print(chr(97))
print(chr(945))
```

The example snippet outputs:

```
a
α
```

Note:

- `chr(ord(x)) == x`
- `ord(chr(x)) == x`

Again, run your own experiments.

Strings as sequences: indexing

We told you before that **Python's strings are sequences**. It's time to show you what that actually means.

Strings aren't lists, but **you can treat them like lists in many particular cases**.

For example, if you want to access any of a string's characters, you can do it using **indexing**, just like in the example below.

```
# Indexing strings

exampleString = 'silly walks'

for ix in range(len(exampleString)):
    print(exampleString[ix], end=' ')

print()
```

Run the program.

Be careful - don't try to pass a string's boundaries - it will cause an exception.

The example output is:

```
s i l l y   w a l k s
```

By the way, negative indices behave as expected, too. Check this yourself.

Strings as sequences: iterating

Iterating through the strings works, too. Look at the example below:

```
# Iterating through a string

exampleString = 'silly walks'

for ch in exampleString:
    print(ch, end=' ')

print()
```

The output is the same as previously. Check.

Slices

Moreover, everything you know about **slices** is still usable.

We've gathered some examples showing how slices work in the string world. Look at the code below, analyze it, and run it.

```
# Slices

alpha = "abdefg"

print(alpha[1:3])
print(alpha[3:])
print(alpha[:3])
print(alpha[3:-2])
print(alpha[-3:4])
print(alpha[::2])
print(alpha[1::2])
```

You won't see anything new in the example, but we want you to be sure that you can explain all the lines of the code.

The code's output is:

```
bd
efg
abd
e
e
adf
beg
```

Now do your own experiments.

The `in` and `not in` operators

The `in` operator shouldn't surprise you when applied to strings - it simply **checks if its left argument (a string) can be found anywhere within the right argument (another string)**.

The result of the check is simply `True` or `False`.

Look at the example program.

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

print("f" in alphabet)
print("F" in alphabet)
print("1" in alphabet)
print("ghi" in alphabet)
print("Xyz" in alphabet)
```

This is how the `in` operator works.

The example output is:

```
True
False
False
True
False
```

As you probably suspect, the `not in` operator is also applicable here.

This is how it works:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

print("f" not in alphabet)
print("F" not in alphabet)
print("1" not in alphabet)
print("ghi" not in alphabet)
print("Xyz" not in alphabet)
```

The example output is:

```
False
True
True
False
True
```

Python strings are immutable

We've also told you that Python's **strings are immutable**. This is a very important feature. What does it mean?

This primarily means that the similarity of strings and lists is limited. Not everything you can do with a list may be done with a string.

The first important difference **doesn't allow you to use the `del` instruction to remove anything from a string**.

The example here won't work:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

del alphabet[0]
```

The only thing you can do with `del` and a string is to **remove the string as a whole**. Try to do it.

Python strings **don't have the `append()` method** - you cannot expand them in any way.

The example below is erroneous:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

alphabet.append("A")
```

with the absence of the `append()` method, **the `insert()` method is illegal**, too:

```
alphabet = "abcdefghijklmnopqrstuvwxyz"

alphabet.insert(0, "A")
```

Operations on strings: continued

Don't think that a string's immutability limits your ability to operate with strings.

The only consequence is that you have to remember about it, and implement your code in a slightly different way - look at the example code.

```
alphabet = "bcdefghijklmnopqrstuvwxy"

alphabet = "a" + alphabet
alphabet = alphabet + "z"

print(alphabet)
```

This form of code is fully acceptable, will work without bending Python's rules, and will bring the full Latin alphabet to your screen:

```
abcdefghijklmnopqrstuvwxyz
```

You may want to ask if **creating a new copy of a string each time you modify its contents worsens the effectiveness of the code**.

Yes, it does. A bit. It's not a problem at all, though.

Operations on strings: `min()`

Now that you understand that strings are sequences, we can show you some less obvious sequence capabilities. We'll present them using strings, but don't forget that lists can adopt the same tricks, too.

Let's start with a function named `min()`.

```
# Demonstrating min() - Example 1
print(min("aAbByYzZ"))


# Demonstrating min() - Examples 2 & 3
t = 'The Knights Who Say "Ni!"'
print('[' + min(t) + ']')

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

The function **finds the minimum element of the sequence passed as an argument**. There is one condition - the sequence (string, list, it doesn't matter) **cannot be empty**, or else you'll get a `ValueError` exception.

The **Example 1** program outputs:

```
A
```

Note: It's an upper-case *A*. Why? Recall the ASCII table - which letters occupy first locations - upper or lower?

We've prepared two more examples to analyze: **Examples 2 & 3**.

As you can see, they present more than just strings. The expected output looks as follows:

```
[ ]
0
```

Note: we've used the square brackets to prevent the space from being overlooked on your screen.

Operations on strings: `max()`

Similarly, a function named `max()` **finds the maximum element of the sequence**.

Look at **Example 1**.

```
1  # Demonstrating max() - Example 1
2  print(max("aAbByYzZ"))
3
4
5  # Demonstrating max() - Examples 2 & 3
6  t = 'The Knights Who Say "Ni!"'
7  print('[' + max(t) + ']')
8
9  t = [0, 1, 2]
10 print(max(t))
```

The example program outputs:

```
z
```

Note: It's a lower-case *z*.

Now let's see the `max()` function applied to the same data as previously. Look at **Examples 2 & 3** in the editor.

The expected output is:

```
[y]
2
```

Carry out your own experiments.

Operations on strings: the `index()` method

The `index()` method (it's a method, not a function) **searches the sequence from the beginning, in order to find the first element of the value specified in its argument**.

```
# Demonstrating the index() method
print("aAbByYzZaA".index("b"))
print("aAbByYzZaA".index("Z"))
print("aAbByYzZaA".index("A"))
```

Note: the element searched for must occur in the sequence - **its absence will cause a ValueError exception**.

The method returns the **index of the first occurrence of the argument** (which means that the lowest possible result is `0`, while the highest is the length of argument decremented by `1`).

Therefore, the example in the editor outputs:

```
2
7
1
```

Operations on strings: the `list()` function

The `list()` function **takes its argument (a string) and creates a new list containing all the string's characters, one per list element**.

Note: it's not strictly a string function - `list()` is able to create a new list from many other entities (e.g., from tuples and dictionaries).

Take a look at the code example.

```
# Demonstrating the list() function
print(list("abcabc"))

# Demonstrating the count() method
print("abcabc".count("b"))
print('abcabc'.count("d"))
```

The example outputs:

```
['a', 'b', 'c', 'a', 'b', 'c']
```

Operations on strings: the `count()` method

The `count()` method **counts all occurrences of the element inside the sequence**. The absence of such elements doesn't cause any problems.

Look at the second example in the editor. Can you guess its output?

It is:

```
2
0
```

Moreover, Python strings have a significant number of methods intended exclusively for processing characters. Don't expect them to work with any other collections. The complete list of is presented here: https://docs.python.org/3.4/library/stdtypes.html#string-methods.

We're going to show you the ones we consider the most useful.