Computer Science I — CSCI-141
Strings — Lecture 1 of 2

09/21/2018

1 Strings of Symbols

Today we will solve some problems that involve a data type known as *strings*.

In computing, a string is often represented by a *sequence of symbols* enclosed in quotation[1] marks. A *sequence* is a collection or composition of symbols arranged in a *specific order*. For strings, the symbols are characters, including punctuation, spaces and numeric characters. The order is the spelling of a word and the structure of a sentence or larger string sequence.

We can create a string variable like this:

```
text = "Hello!"
```

And we can visualize this assignment this way: *text* $\longmapsto$

H	e	l	l	o	!

The variable name refers to the start character of the sequence, and each character follows its predecessor in memory. Individually each character is usually written inside quotes. For example, we might write `"e"` and Python printed output might show `'e'`.

1.1 Problem 1: Sentence and Word Length

Given a sentence, we want to know its length and the length of the longest word in that sentence. To answer that, we need to define the meanings of "sentence" and "word".

For example, the sentence "This is funny." has three words: "This", "is", "funny", and one punctuation character, ".". Except for the last word and the period, the words are separated by a space character.

While it is possible to separate out punctuation that is adjacent to an actual English word, we will keep things simple and count adjacent characters and adjacent punctuation as a single word. That means we will treat the sequence "funny." as a 6-character word.

We will define a *sentence* as any string of symbols, and a *word* as any sequence of symbols not containing what's known as *whitespace*, which are symbols whose appearance is not visible. For this problem, whitespace includes the space character and two other symbols: the **TAB** character (`'\t'`), and the **newline** character (`'\n'`).

1.2 Problem 2: Is the reverse spelling of a word a valid word?

Some word puzzles play with the order of the characters. One game is to find words that are other words when spelled backwards. One of the names of this game is "semordnilap"[2],

1. Python allows either single or double quotes.
2. Another name for this is "reversegram". See `https://en.m.wiktionary.org/wiki/semordnilap`.

which is the reverse spelling of “palindromes” – words spelled the same backwards or forwards.

In order to determine, if a word is a semordnilap, it would be nice to have a function that returns the reverse of a given word.

2 Learning to Process Strings in the Python Language

In Python a *string* is an instance of a sequence of characters written explicitly between two matching single or double quote symbols. Here are some examples shown in the Python interpreter console.

```
>>> "Hello World"
'Hello World'
>>> ''
''
>>> "a"
'a'
>>> 'abc'
'abc'
```

The empty string, “”, is the shortest valid string, and its length is 0.

2.1 Indexing, Slicing, and Concatenating

It is possible to access parts within a string sequence.

One way to do that is by **indexing**, which is written as st`[`n`]`, where st is an expression that symbolizes a string and n is an expression that symbolizes an integer. The value of st`[`n`]` is the string that is the character at position n in string st.

```
>>> st = 'Hello World'
>>> st[0]
'H'
>>> st[1]
'e'
>>> st[2]
'l'
>>> st[5]
' '
```

It is important to note that **indexing is zero-based**, which means that the first symbol in a non-empty string is at index `0`. It is an error, if the index value is out of range.

```
>>> st = ''
>>> st[0]
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
```

We can assign string values to variable names, but it is not possible to change the symbols within a string using the variable. We say that **strings are immutable**, which means parts of strings cannot be changed using the assignment operator. If we assign a new value to a string variable, we replace its old value.

```
>>> st = 'abc'
>>> st[0]
'a'
>>> st[0] = 'z'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

There is another way to access parts of a string. **Slicing creates a new string** that is a copy of the substring of the original, sliced string. The *slice* is a copy of some substring of the original.

A slice is written st`[`m`:`n`]`, where st is an expression that symbolizes a string, and m and n are expressions that represent integers. The integer expressions are both optional.

The value of st`[`m`:`n`]` is the substring that starts with the character at position m and continues up to, *but not including*, the character at position n. If m is omitted, the starting character is the first character, and, if n is omitted, it is the substring that goes to the end of the string.

```
>>> st[1:4]
'ell'
>>> st[:5]
'Hello'
>>> st[1:]
'ello World'
```

It is also possible to **concatenate** strings so that the result is a new string composed of the first string followed by the second string. In Python, string concatenation is written using a plus-sign (+).

```
>>> 'Hello' + ' World'
'Hello World'
>>> 'a' + 'bc'
'abc'
>>> 'ab' + 'c'
'abc'
>>> 'a' + 'b' + 'c'
'abc'
```

2.2 Iteration over Strings with a for Loop

Python provides a statement that makes a variable *step through a string* symbol-by-symbol: the `for` loop.

There are several *patterns* of the `for` loop, and here is the first one for strings:

```
# this 'for loop' processes each character one at a time
for ch in st:
    # This block of statements that use ch, where
    # ch is the 'next' character in the string st
    # for each subsequent cycle of for-loop execution.
    # ...
# statements after the for loop ...
```

To describe this particular pattern, we say that *ch* is a **loop variable** and *st* is an expression that refers to a string[3]. This loop *iterates over*, or steps through, the string executing the statements in the block once for each character in the string. At the first iteration, *ch* refers to the first character in the string; at the second iteration, *ch* refers to the second character in the string; and so on. Below is an example written and executed in the interpreter console.

```
>>> for ch in 'abc':
...     print(ch)
...
a
b
c
```

2.3 Generating Index Values with `range`

In some situations it is necessary to access the elements of a string `st` by their index. We might need to iterate over the range of values using their position `0`, `1`, `2`,..., `len(st) - 1`, where the `len(st)` is a *built-in function* that returns the length of the string `st`.

This leads to another pattern of `for` loop in Python: the "*index loop*".

To illustrate, let us say that we wanted to output all characters in the string `st` except for the one found at the third position. One way to do it would be by iterating over *st* and skipping the element with the index `2`.

```
>>> st = 'abcd'
>>> for i in range(0, len(st)):
...     if not i == 2:
...             print(st[i])
...
a
```

3. The `for` loop can iterate through sequences other than strings.

```
b
d
>>>
```

We could get the same result by creating a new string `st = st[0:2] + st[3:]`, and then looping over its elements.

The function `range(a, b)` produces the integers between `a` (included) and `b` (excluded) to the `for` loop. For instance, the following code calculates the factorial of `n`:

```
def fact( num):
    """
    fact : Non-negative-integer -> Positive-integer
    Calculate the factorial of num and return the result.
    """
    val = 1
    for i in range( 1, num+1):
        val = val * i
    return val
```

If we wrote `for i in range(2, 5):` then the iteration would generate the sequence 2, 3, 4, a loop of three cycles binding i to each value in turn. In mathematical terms, the function `range(a, b)` where $a < b$, produces the integer sequence $[a, a + 1, \ldots b)$ as the loop iterates.

3 Solutions

3.1 Solution 1: Computing the Length of the longest word

3.1.1 Length of a String "Sentence"

Although the Python function `len` does return a string's length, we will write our own version to compute the length of a string.

```
def length( st):
    """
    length : String -> Number
    length returns the length of st.
    """
    size = 0
    for ch in st:
        size = 1 + size
    return size
```

3.1.2 Length of a Word within a String Sentence

While we can use `length` to calculate the length of a string, it will count whitespace too. We will write a `longest` function to find the longest substring of non-whitespace characters. The function needs to:

1. Initialize a result value.
2. Initialize a size value for the size of the current word.
3. Iterate over the string and:
4. (a) If the next character is whitespace, reset the size to 0.
   Otherwise, increment the size of the word.
   (b) If the new, longer substring size is greater than the result, update the result value.
5. Return the result.

3.1.3 Execution Timeline

When developing an iterative function it is useful to create a timeline of execution to validate the algorithm. Time progresses from left to right, or from top to bottom, in a time table.

A timeline for tracing the for loop in the call `longest( 'Hi there')` might go something like the picture below. Each column represents a value at the *start of the* `for` *loop*.

Assignments	Time ⟶								
size = 0	1	2	0	1	2	3	4	5	5
result = 0	1	2	2	2	2	3	4	5	5
ch = 'H'	'i'	' '	't'	'h'	'e'	'r'	'e'	''	undefined after for

3.1.4 Complexity

Processing string data often involves stepping through the string from beginning to end. The `for` loop runs to the end of the string in the worst case, and consequently many string processing functions are processed in linear time over N, where N is the length of the string.

If we simply want to access an arbitrary position within a string, the indexing operation is constant time because the number of operations required to compute the position and return the value is constant and never changes for any value of i in the expression $st[i]$.

3.1.5 Testing and Python Modules

For the `length` function, we need to test that the string is correct for empty and non-empty strings, up to some small size. In our implementation, we use these cases: `''`, `'a'`, `'ab'`, and `'abc'`.

For the `longest` function, we need to test that it produces the correct results for the empty string and for the cases in which the longest word is at the beginning, middle, or end of the sentence.

The `lengths.py` is a simple example of a utility function module. When we write functions to process a data type, we might want to reuse those functions later. Because the `longest` function may be useful for other programs, we have put the string functions into their own *module*.

Similarly, we create a *test module* file containing the test functions for our function module. This will minimize the size of the `lengths.py` module by keeping the test functions separate from the implementation. The `test_lengths.py` is a simple example of a test module.

3.1.6 Longest Word Solution

The `find_lengths.py` file is a solution to the first problem. It prompts for a sentence, and calls `longest` to get the length of the longest sequence of non-whitespace characters.

3.2 Solution 2: Computing the Reverse of a String

We need a function that takes the characters of a word and creates a string in reverse. The idea is simple: take each character and put it in front of the reverse of the characters that are after it.

1. Initialize the result.
2. Iterate over the string to:
   (a) Concatenate the next character onto the front of the result
3. Return the result.

Then the main program simply prompts for a word, passes the input to the reverse function, and prints the return value from the function.

3.2.1 'semordnilap' Solution

For this problem, we kept everything in one file: `semordnilap.py`. This contains two versions of a reverse function, a test function and a main function.

For the `reverse` and the `reverse_by_index` function, we can use the same tests if we can tell the function which function to execute. The test function shows that you can *pass a function as an argument*, and then call that function using the parameter name. We test that the reverse functions produce correct results for the empty string as well as several non-empty strings. For simplicity, we will manually examine the resulting word to determine if the result is in fact a valid English word.

4 Implementations

Complete Python implementations and tests for solution 1 are found in `find_lengths.py`, `lengths.py` and `test_lengths.py`. See `semordnilap.py` for solution 2, the reversegram program.