Lesson 6: Strings

Mandoye Ndoye, PhD

October 16, 2017

Department of Electrical Engineering
Tuskegee University

A string is a sequence

- A string is a sequence of characters: you can access the characters one at a time with the bracket operator []

```
>>> fruit = 'banana'
>>> letter = fruit[0]
>>> print letter
'b'
```

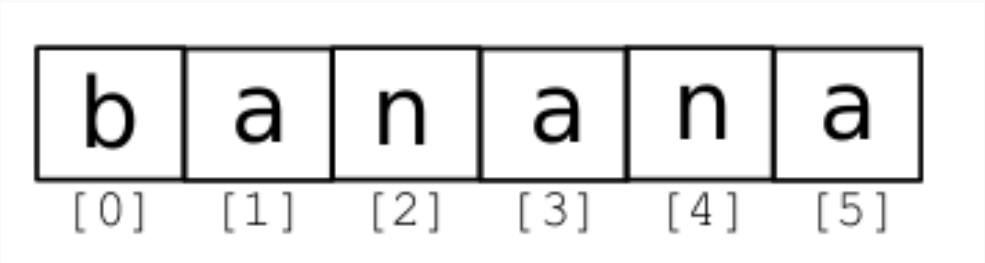
- The second statement above extracts the first character (at index 0) from string variable `fruit` and assigns it to the variable `letter`

- The expression in the brackets is called an **index**, and indicates which character in the sequence you want to extract

A string is a sequence

- In Python, the index is an offset from the beginning of the string, and the offset of the first letter is zero

- Thus, `b` is the 0-th letter (zero-eth) of the string `'banana'`, `a` is the 1-th letter (one-eth), `n` is the 2-th (two-eth) letter, and so on



- Any expression that evaluates to an integer can be used as a valid index

```
>>> fruit = 'banana'; n = 1; alpha = 2
>>> letter = fruit[2*n + alpha]; print letter
'n'
```

Getting the length of a string using `len`

- `len` is a function that returns the number of characters in a string:

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

- A common mistake in trying to get the last letter of a string is:

```
>>> length = len(fruit); last = fruit[length]
IndexError: string index out of range
```

- To get the last character, subtract 1 from the length of the string:

```
>>> last = fruit[length-1]; print last
'a'
```

- A simpler notation is to use negative indices to count backward from the end of the string: `fruit[-1]` returns the last letter, `fruit[-2]` yields the second to last, and so on

Traversal through a string with a loop

- Many computations a pattern of processing called a **traversal**, where a string is processed one character at a time as follows
  1. Start at the beginning of the string
  2. Select each character in turn
  3. do something related to the current character
  4. Repeat steps 2 and 3 until the end of the string

- A traversal can be implemented with a `while` loop:

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

- The loop traverses the string and displays each letter on a line by itself. The loop condition is false once index is equal `len(fruit)` because the last character of the string has index `len(fruit)-1`

Traversal through a string with a loop

- Another way to implement a traversal is with a `for` loop:

```
for char in fruit:
    print char
```

- Each time through the loop, the next character in the string is assigned to the variable char. The loop continues until no characters are visited.

- **Exercise:** *Write a* `while` *loop that starts at the last character in the string and works its way backwards to the first character in the string, printing each letter on a separate line, except backwards*

- **Exercise:** *Repeat the previous exercise using a* `for` *loop*

String slices

- Segment of a string is called a **slice**, and can be selected using the `[n:m]` operator

```
>>> s = 'Monty Python'; print s[6:12]
'Python'
```

- The slice operator `[n:m]` returns the part of the string from the n-th character to the $(m-1)$-th character

- If the "start" argument `n` is omitted, the slice starts at the beginning of the string:

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

- If the "stop" argument `m` is omitted, the slice goes to the end of the string:

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

String slices

- If the "stop" argument is beyond the end of the string, it just stops at the end

```
>>> fruit = 'banana'; fruit[3:100]
'ana'
```

- If the "stop" argument `n` is greater than or equal to the "stop" argument `m`, the result is an **empty string**

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

- An empty string is represented by two matching quotation marks and has length zero

- **Exercise:** *Given that the variable* `fruit` *is a string, what is* `fruit[:]`*?*

String slices

- The slice operator support an optional third "step" or "stride" argument, which is can be very useful in numerical computing

```
>>> import string
>>> aString = string.lowercase
>>> print aString
abcdefghijklmnopqrstuvwxyz
>>> aString[3:20:4]
'dhlpt'
```

- If the "step" argument is omitted, it takes the default value of 1

```
>>> aString = string.punctuation
>>> print aString
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> aString[10:20:]
'+,-./:;<=>'
```

String slices

- When both the "start" and "stop" arguments are omitted in an extended slice, its behavior depends on the sign of the "step" argument

- If "step" argument is a positive integer then, as expected, the string is traversed from its start all the way to its end

```
>>> aString = string.uppercase
>>> print aString
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> aString[::2]
'ACEGIKMOQSUWY'
```

- If "step" argument is a negative integer, the string is traversed in reverse order from the end to its start

```
>>> aString[::-1]
'ZYXWVUTSRQPONMLKJIHGFEDCBA'
```

Strings are immutable

- You cannot change the value of an existing string

```
>>> greeting = 'Zood morning!'
>>> greeting[0] = 'G'
TypeError: object does not support item assignment
```

- Can however create a new string that is a variation on the original

```
>>> greeting = 'Zood morning!'
>>> new_greeting = 'G' + greeting[1:]
>>> print new_greeting
Good morning!
```

- The new string could have the same name as the original

```
>>> greeting = 'Zood morning!'
>>> greeting = 'G' + greeting[1:]
>>> print greeting
Good morning!
```

Using the + operator to concatenate strings

- Apply + to strings to perform "concatenation"

```
>>> a = 'Tuskegee'; b = a + 'University'
>>> print b
TuskegeeUniversity
>>> c = a + ' ' + 'University'
>>> print c
Tuskegee University
```

- Both operands must be strings: the following command gives an error

```
print 2 + 'elephants'
```

- Use `str()` to convert variables or data that are not strings

```
>>> print 'These ' + str(2) + ' elephants will cost ' \
            + str(1.24) + ' dollars'
These 2 elephants will cost 1.24 dollars
```

Using the * operator to repeat strings

- Apply * to a string and a (positive) integer to perform *repetition*

```
>>> a = '!'; b = 4 * '!'
>>> print b
!!!!
>>> print 'Go' + '\t' + 'Tuskegee' + 2*b
Go    Tuskegee!!!!!!!!
```

- The integer operand can be any expression that evaluate to a integer

```
>>> m = 2; n = 4
>>> print  'Go' * (2*m + n)
'GoGoGoGoGoGoGoGo'
```

- If the integer is zero or negative , an empty string is returned

```
>>> print  0 * 'Go'
''
>>> print  'Go' * (-3)
''
```

The `in` operator

- The operator `in` is used to check to see if one string is in another string

```
>>> 'u' in 'Tuskegee'
True
>>> 'sweet' in 'Sweet Home Alabama'
False
```

- The `in` operator is a boolean (i.e., returns either `True` or `False`), and thus can be used in `if`, `while` or `for` conditions

```
>>> letter = 'a'
>>> fruit = 'avocado'
>>> if letter in fruit:
        print letter, 'is in', fruit

a is in avocado
```

Looping and counting

- The `in` operator can be used to visit each letter in a string and counts the number of times the loop encounters a given character

```
letter = 'a'
word = 'banana'
occurrence_of_letter = 0
for item in word:
    if item == letter:
        occurrence_of_letter = occurrence_of_letter + 1
print letter, 'occurs', occurrence_of_letter, 'times'
```

- **Exercise:** *Encapsulate this code in a function named count, and generalize it so that it accepts the string and the letter as arguments*

String comparison

- Comparison operators work on strings

```
>>> word = 'apple'
>>> print word == 'banana'
False
```

- Comparison operations are useful for alphabetically ordering words.

- However, in Python ordering, uppercase letters come before the lowercase letters.

```
>>> word1 = 'apple'; word2 = 'Apple'
>>> print word 1 > word2
True
```

- This problem can be addressed by converting strings to a standard format, such as all lowercase, before performing an alphabetical comparison

String comparison

- To understand Python's way of ordering characters, use the `ord()` function to convert a character to its corresponding decimal ASCII value

```
>>> print ord('A'),':',ord('Z'),':',ord('a'),':',ord('z)
65 : 90 : 97 : 122
```

- Use the `chr()` function get a ASCII character from its corresponding decimal (or hexadecimal) value

```
>>> print chr(90), ':' , chr(0x5A)
'Z' : 'Z'
```

- Use the `hex()` function to convert decimal to HEX, and the `int()` function to convert from HEX to decimal

```
>>> print hex(90), ':', int(0x5a)
0x5a : 90
```

String methods

- Strings are an example of Python **objects**

- An object contains
  - data (e.g., the actual string itself in this case)
  - constants that are associated to the object
  - methods that are functions built into the object and are available to any **instance** of the object

- Python has a function called `dir()` that lists the methods available for an object

  The `type()` function shows the type of an object whereas the `dir()` function shows available methods

String methods

- Available methods for strings are listed below

```
>>> some_string = 'hello'
>>> dir(some_string)
[.....................................................,
'capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
```

String methods

- You can use `help()` to get some simple documentation on a method:

```
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> string

        Return a copy of the string S with only
        its first character capitalized.
```

- A better source of documentation for string methods: `docs.python.org/library/string.html`

- Although the syntax is different, calling a **method** is similar to calling a function: A method takes arguments and returns a value

String methods

- A method is called by appending its name to the variable name (or directly to the data) using the period as a delimiter

- Example: Applying the method `upper` to a string

```
>>> my_string = 'james'
>>> uppercase_string = my_string.upper()
>>> print uppercase_string
'JAMES'
```

- This dot notation specifies the name of the method, `upper`, and the string to apply the method to. The empty parentheses indicate that the method takes no argument in this **invocation**[1]

[1] A method call is referred to an invocation; in the above example, we are invoking the method `upper` on the string `my_string`

String methods: Is every character ...

- Methods that whether or not every character in a string S possesses a certain property

Syntax	Action
S.islower()	True if every character is lowercase (a-z)
S.isupper()	True if every character is uppercase (A-Z)
S.isalpha()	True if every character is alphabetic (A-Z,a-z)
S.isdigit()	True if every character is numeric (0-9)
S.isalnum()	True if every character is alphanumeric (A-Z,a-z,0-9)
S.isspace()	True if every character in whitespace (' ', '\n', '\t', '\r')

- All the above methods return `False` if the string S is empty

String methods: Is every character ...

- Example usages of methods testing every character for a property

```
>>> 'President Obama'.isalnum()
False
>>> 'apple'.islower()
True
>>> '15:32'.isdigit()
False
>>> '\n\t  \r  \n'.isspace()
True
>>> 'Hello'.isupper()
False
>>> ''.isspace()
False
```

String methods: Testing for a starting or ending substring

- Methods to test whether a string S starts or ends with a given substring

Syntax	Action
S.startswith(substring[,**start**[,**end**]])	True if S starts with *substring*
S.endswith(substring[,**start**[,**end**]])	True if S ends with *substring*

- **start**: *optional parameter/index to set start of search region*
- **end**: *optional parameter/index to set end of the search region*
- To simply test for the presence of a substring within a string, use `in` operator

String methods: Testing for a starting or ending substring

- Example usages of methods `.startswith` and `.endswith`

```
>>> 'www.tuskegee.edu'.startswith('www')
True
>>> 'www.tuskegee.edu'.startswith('WWW')
False
>>> 'www.tuskegee.edu'.endswith('edu')
True
>>> 'www.tuskegee.edu'.endswith('com')
False
```

- To find substrings anywhere inside a string:

```
>>> 'www' in "our hbcu's webpage is www.tuskegee.edu"
True
```

String methods: Locating a substring

- String methods that return the location of a substring

Syntax	Action
S.index(substring [,start [,end]])	Return the index of first (leftmost) occurrence of substring; if found *if not found, raise ValueError*
S.rindex(substring [,start [,end]])	Return the index of last (rightmost) occurrence of substring; if found *if not found, raise ValueError*
S.find(substring [,start [,end]])	Return the index of first (leftmost) occurrence of substring; if found if not found, return -1
S.rfind(substring [,start [,end]])	Return the index of last (rightmost) occurrence of substring; if found if not found, return -1

String methods: Locating a substring

- Example usages of `.index()` and `.rindex()`

```
>>> 'greatness or madness'.index('ness',6)
16
>>> 'greatness or madness'.rindex('ness')
16
```

- Example usages of `.find()` and `.rfind()`

```
>>> 'greatness or madness'.find('ness')
5
>>> 'greatness or madness'.rfind('ness',6,12)
-1
```

- Key difference: If the substring is not found, `.find()` and `.rfind()` return -1 where as `.index()` and `.rindex()` produce an error

String Methods: Counting substrings

- A method that returns the number of occurrences of a given substring:

Syntax	Action
S.count(substring[, **start**[, **end**]])	Count occurrences of substring

- Examples of usage

```
>>> 'Traveling is learning'.count('ing')
2
>>> 'Traveling is learning'.count('ing',10)
1
>>> 'Traveling is learning'.count('ing',8,15)
0
```

String Methods: Case manipulators

- String methods that manipulate letter cases:

Syntax	Action
S.lower()	Convert S to lowercase
S.upper()	Convert S to uppercase
S.capitalize()	Convert first character of S to uppercase
S.title()	Convert first character of every word of S to uppercase
S.swapcase()	Convert uppercase to lowercase and vice versa

- Examples of usage:

```
>>> 'Do not disturb'.upper()
'DO NOT DISTURB'
>>> 'green eggs and ham'.capitalize()
'Green eggs and ham'
>>> 'green eggs and ham'.title()
'Green Eggs And Ham'
```

String Methods: Transforming a string

- Methods that replace or remove parts of a string

Syntax	Action
S.replace (old, new[, **count**])	Replace all occurrences of the substring *old* with the substring *new*
S.lstrip([**char**])	Remove **char** from the front (left) of S; whitespace is the default argument
S.rstrip([**char**])	Remove **char** from the end (right) of S; whitespace is the default argument
S.strip([**char**])	Remove **char** at the front and end of S; whitespace is the default argument

- In `S.replace`, if the optional argument **count** is given, only the first **count** occurrences are replaced

String methods: Transforming a string

- Example usages of `.replace()`

```
>>> mary = 'Mary had a lamb'; mary.replace('a', 'xx')
'Mxxry hxxd xx lxxmb'
>>> chom = 'Colorless green ideas sleep furiously';
chom.replace(' ', '')
'Colorlessgreenideassleepfuriously'
>>> chom.replace('ee', '')
'Colorless grn ideas slp furiously'
```

- Example usages of `.lstrip()`, `.rstrip()` and `.strip()`

```
>>> foo = '\t hello world\n \n'
>>> print foo
     hello world
>>> print foo.lstrip()
'hello world'
```