

CMPSC-122: Intermediate Programming

Spring 2018

Department of Computer Science & Engineering
The Pennsylvania State University

1. Strings

Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Creating strings is as simple as assigning a value to a variable. Since strings are made up of smaller pieces (characters), they belong to the *compound data types*, where we can treat strings as a single thing or we can just access its parts.

```
message = "Hello, World!"  
upper_message = message.upper()  
print(upper_message)    #'HELLO, WORLD!'
```

A string instance has its own attributes and methods. *upper* is a method that can be invoked on any string object to create a new string, in which all the characters are in uppercase (The original string *message* remains unchanged.)

```
fruit = "strawberry"  
letter = fruit[3]  
print(letter)          #a
```

The expression *fruit[3]* selects character number 3 from *fruit*, and creates a new string containing just this one character. The variable *letter* refers to the result. (Remember that the indexing starts counting from zero)



Length

The *len* function, when applied to a string, returns the number of characters in a string:

```
fruit = "strawberry"  
len(fruit)    #10
```

To get the last letter of a string, you might be tempted to try something like this:

```
string_size = len(fruit)  
last = fruit[string_size]
```

That won't work. It causes the runtime error `IndexError: string index out of range`. The reason is that there is no character at index position 10 in "strawberry". Because we start counting at zero, the ten indexes are numbered 0 to 9. To get the last character, we have to subtract 1 from the length of fruit:

```
string_size = len(fruit)  
last = fruit[string_size-1]
```

Alternatively, we can use negative indices, which count backward from the end of the string. The expression *fruit[-1]* yields the last letter, *fruit[-2]* yields the second to last, and so on.

1.1 Traversal, the *for* and *while* loops

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a **traversal**. We can encode a traversal with a *while* or a *for* loop:

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



This loop traverses the string and displays each letter on a line by itself. The loop condition is $x < \text{len}(\text{fruit})$, so when x is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index $\text{len}(\text{fruit})-1$, which is the last character in the string.

```
fruit = "strawberry"
for x in fruit:
    print(x)
```

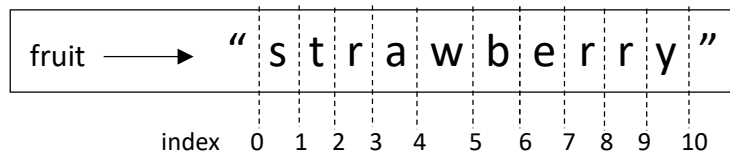


Each time through the loop, the next character in the string is assigned to the variable x . The loop continues until no characters are left.

The examples above show the expressive power that the *for* loop gives us compared to the *while* loop when traversing a string.

1.2 Slices

A substring of a string is obtained by taking a **slice**. The operator $[n:m]$ returns the part of the string from the n 'th character to the m 'th character, including the first but excluding the last. This behavior makes sense if you imagine the indices pointing between the characters:



If you imagine the picture above as a piece of paper, the slice operator $[n:m]$ copies out the part of the paper between the n and m positions. Provided m and n are both within the bounds of the string, and your result will be of length $(m-n)$. If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice extends to the end of the string. Similarly, if you provide value for n that is bigger than the length of the string, the slice will take all the values up to the end and it won't give an "out of range" error like the normal indexing operation does. Slicing applies to list too!

```
movie = "The Silence of the Lambs"
print(len(movie))      # 24

print(movie[0:11])     # The Silence
print(movie[12:14])    # of
print(movie[15:24])    # the Lambs
print(movie[:])         # The Silence of the Lambs
print(movie[4:])        # Silence of the Lambs
```



String Comparison

The comparison operators work on strings. To see if two strings are equal:

```
word = "apple"
if word == "apple":
    print("Yes, we have no apples!")
```

Other comparison operations are useful for putting words in *lexicographical* order:

```
word = "apple"
if word < "banana":
    print("Your word, " + word + ", comes before banana")
elif word > "banana":
    print("Your word, " + word + ", comes after banana")
else:
    print("Yes, we have no apples!")
```

However, all the uppercase letters come before all the lowercase letters. As a result, if *word*=Zucchini:

```
Your word, Zucchini, comes before banana
```

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison.

1.3 Strings are immutable

It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string:

```
greeting = "Hello, World!"
greeting[0] = 'J'
print(greeting)
```

Instead of producing the output 'Jello, world!', this code produces the runtime error `TypeError: 'str' object does not support item assignment`. Strings are immutable, which means you cannot change an existing string. The best you can do is create a new string that is a variation on the original:

```
greeting = "Hello, world!"
new_greeting = "J" + greeting[1:]
print(new_greeting)
```

The solution here is to concatenate a new first letter onto a slice of *greeting*. Remember, this operation has no effect on the original string!!

1.4 The *in* and *not in* operators

The *in* operator tests for membership. When both of the arguments to *in* are strings, *in* checks whether the left argument is a substring of the right argument. Note that a string is a substring of itself, and the empty string is a substring of any other string. The *not in* operator returns the logical opposite results of *in*:

```
"p" in "apple"      #True
"i" in "apple"      #False
"ap" in "apple"     #True
"pa" in "apple"     #False
"apple" in "apple"  #True
"" in "apple"       #True
"x" not in "apple"  #True
```

1.5 String methods: *find()*, *split()* and *format()*

There are some of the built-in methods that allow us to easily make modifications to strings in Python. Usually we would prefer to use the methods that Python provides rather than reinvent our own equivalents.

1.5.1 The *find* method

It determines if *string1* occurs in *string2*, or in a substring of *string2* if a starting and ending index are given:

```
string1.find(string1, beg=0, end=len(string2))
```


Parameters

str -- This specifies the string to be searched
beg -- This is the starting index, by default its 0
end -- This is the ending index, by default its equal to the length of the string

Return Value

Index if found and -1 otherwise.

```
fruit="banana "  
fruit.find("nan")      #2  
fruit.find("na",3)     #4  
fruit.find("na",1,2)   #-1
```



1.5.2 The *split* method

One of the most useful methods on strings is the *split* method: it splits a single multi-word *string* into a *list* of individual words, removing all the whitespace between them. (Whitespace means any tabs, newlines, or spaces.) This allows us to read input as a single string, and split it into words.

```
message = "We are Penn State!"  
words = message.split()  
print(words)      #['We', 'are', 'Penn', 'State!']
```

1.5.3 The *format* method

This method performs a string formatting operation. The string on which this method is called can contain literal text or replacement fields delimited by braces {}. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument.

The template string contains place holders, ... {0} ... {1} ... {2} ... etc. The *format* method substitutes its arguments into the place holders. The numbers in the place holders are indexes that determine which argument gets substituted. Each of the replacement fields can also contain a format specification (introduced by the : symbol). This modifies how the substitutions are made into the template, and can control things like:

- whether the field is aligned to the left <, center ^, or right >
- the width allocated to the field within the result string
- the type of conversion (if the type conversion is a float, you can also specify how many decimal places are wanted)

A few simple and common examples that should be enough for most needs are:

```
s1 = "Her name is {0}!".format("Kate")  
print(s1)                        # Her name is Kate!  
  
name = "Isaac"  
age = 35  
s2 = "I am {1} and I am {0} years old.".format(age, name)  
print(s2)                        # I am Isaac and I am 35 years old
```

```

n1 = 4
n2 = 5
s3 = "2**10 = {0} and {1} * {2} = {3:f}".format(2**10, n1, n2, n1 * n2)
print(s3)
# 2**10 = 1024 and 4 * 5 = 20.000000

name1 = "James"
name2 = "Charlie"
name3 = "Valerie"
print("Pi to three decimal places is {0:.3f}".format(3.1415926))
# Pi to three decimal places is 3.142

print("123456789 123456789 123456789 123456789 123456789 123456789")
# 123456789 123456789 123456789 123456789 123456789 123456789

print("|||{0:<16}|||{1:^16}|||{2:>16}|||Born in {3}|||".format(name1,name2,name3,1986))
# |||James          |||   Charlie   |||          Valerie|||Born in 1986|||

print("The decimal value {0} converts to hex value {0:x}".format(1234567))
# The decimal value 1234567 converts to hex value 12d687

```

If you need to do anything more than what was shown in the examples above, you can visit <https://docs.python.org/3/library/string.html#formatexamples> and read more details and examples of the `str.format()` syntax.

Appendix 1. Built-in String Methods

Python includes the following built-in methods to manipulate strings:

capitalize() - Returns the string with first letter capitalized and the rest lowercased.

casefold() - Returns a lowercase string, generally used for caseless matching. This is more aggressive than the lower() method.

center() - Center the string within the specified width with optional fill character.

count() - Count the non-overlapping occurrence of supplied substring in the string.

encode() - Return the encoded version of the string as a bytes object.

endswith() - Returns true if the string ends with the supplied substring.

expandtabs() - Return a string where all the tab characters are replaced by the supplied number of spaces.

find() - Return the index of the first occurrence of supplied substring in the string. Return -1 if not found.

format() - Format the given string.

format_map() - Format the given string.

index() - Return the index of the first occurrence of supplied substring in the string. Raise ValueError if not found.

isalnum() - Return true if the string is non-empty and all characters are alphanumeric.

isalpha() - Return true if the string is non-empty and all characters are alphabetic.

isdecimal() - Return true if the string is non-empty and all characters are decimal characters.

isdigit() - Return true if the string is non-empty and all characters are digits.

isidentifier() - Return true if the string is a valid identifier.

islower() - Return true if the string has all lowercased characters and at least one is cased character.

isnumeric() - Return true if the string is non-empty and all characters are numeric.

isprintable() - Return true if the string is empty or all characters are printable.

isspace() - Return true if the string is non-empty and all characters are whitespaces.

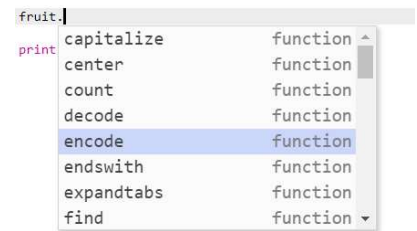
istitle() - Return true if the string is non-empty and title cased.

isupper() - Return true if the string has all uppercased characters and at least one is cased character.

join() - Concatenate strings in the provided iterable with separator between them being the string providing this method.

ljust() - Left justify the string in the provided width with optional fill characters.

lower() - Return a copy of all lowercased string.



lstrip() - Return a string with provided leading characters removed.

maketrans() - Return a translation table.

partition() - Partition the string at first occurrence of substring (separator) and return a 3-tuple with part before separator, the separator and part after separator.

replace() - Replace all old substrings with new substrings.

rfind() - Return the index of the last occurrence of supplied substring in the string. Return -1 if not found.

rindex() - Return the index of the last occurrence of supplied substring in the string. Raise ValueError if not found.

rjust() - Right justify the string in the provided width with optional fill characters.

rpartition() - Partition the string at last occurrence of substring (separator) and return a 3-tuple with part before separator, the separator and part after separator.

rsplit() - Return a list of words delimited by the provided substring. If maximum number of split is specified, it is done from the right.

rstrip() - Return a string with provided trailing characters removed.

split() - Return a list of words delimited by the provided substring. If maximum number of split is specified, it is done from the left.

splitlines() - Return a list of lines in the string.

startswith() - Return true if the string starts with the provided substring.

strip() - Return a string with provided leading and trailing characters removed.

swapcase() - Return a string with lowercase characters converted to uppercase and vice versa.

title() - Return a title (first character of each word capitalized, others lowercased) cased string.

translate() - Return a copy of string that has been mapped according to the provided map.

upper() - Return a copy of all uppercased string.

zfill() - Return a numeric string left filled with zeros in the provided width.

References

<https://docs.python.org/3/library/stdtypes.html>

<https://www.tutorialspoint.com/python/index.htm>