
# 01-3 Python Strings

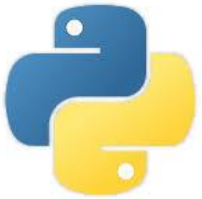
CSI 500

Spring 2018

Note: course material adopted loosely from :
Downey, Allen B. *Python for software design: how to think like a computer scientist*. Cambridge University Press, 2009.
http://greenteapress.com/wp/think-python/

# A string is a sequence

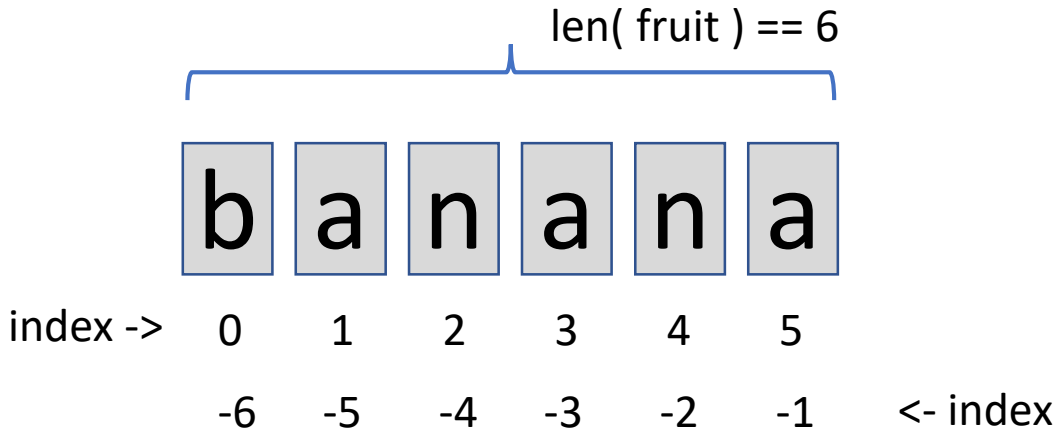
- A string is a sequence of characters
  - Individual elements of the string are accessed using the bracket operator
  - the expression in the bracket is called the "index"
  - string indexes start at 0, not 1 as you might expect
  - index may be an integer or an expression; will fail if non-integer



```
>>> fruit = 'banana'
>>> letter = fruit[ 1 ]
>>> letter
'a'             # huh? shouldn't this be 'b' ???

>>> offset = 2
>>> fruit[ offset + 1 ]
'a'
>>> bad_offset = 1.5
>>> fruit[ bad_offset ]
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    fruit[ bad_offset ]
TypeError: string indices must be integers
>>>
```

# len

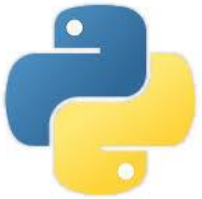
- the built-in len function returns the length of a string
- note: this is one more than the index of the last character in the string
- it is Pythonic to access the end of a string using negative indexes




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

# this won't work... !
>>> length = len( fruit )
>>> length
6
>>> last = fruit[ length ]
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    last = fruit[ length ]
IndexError: string index out of range
```

# Traversing strings with loops

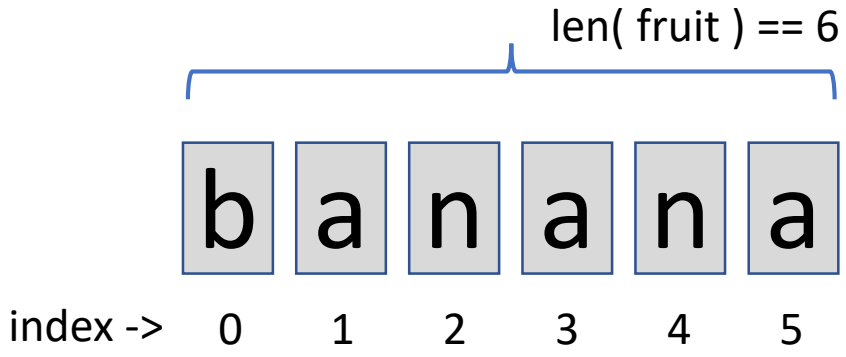
- A while loop can be used with len() to traverse a string
  - remember to increment the loop control variable !
- A for loop can also be used
  - does not require an explicit loop control variable



```
>>> index = 0
>>> while index < len( fruit ):
        letter = fruit[ index ]
        print( letter )
        index = index + 1
b
a
n
a
n
a
>>> for letter in fruit:
        print( letter )
b
a
n
a
n
a
```

# String slices

- A slice is a segment of a string
- The [n:m] operator is used to create slices
  - n is the first index
  - m is one more than the last index
- [ :m ] starts at index 0 and goes to m-1
- [ n: ] starts at n and goes to len( ) - 1

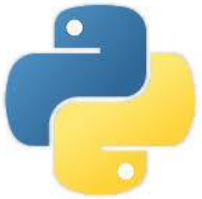



```
>>> s = 'Monty Python'
>>> s[ 0:5 ]
'Monty'
>>> s[ 6:12 ]
'Python'

>>> fruit = 'banana'
>>> fruit[ :3 ]
'ban'
>>> fruit [ 3: ]
'ana'

>>> # what will this do?
>>> fruit[ : ]
```

# Strings are immutable

- You can't update elements of a string
  - Python will return an error
- You can create a new string based on an existing string

```
>>> greeting = 'Hello world!'
>>> greeting[ 0 ] = 'J'
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    greeting[0] = 'J'
TypeError: 'str' object does not support item assignment

# make a new string from an old one
>>> new_greeting = 'J' + greeting[ 1: ]
>>> new_greeting
'Jello world!'
```



# Searching

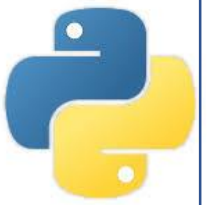
- We've seen how to get the character at a given index using the [] operator
- Consider the inverse: how to get the index of a specified character?
  - start at index == 0
  - search thru word one char at a time
  - if found, exit and return index
  - if not found, return -1

```
>>> def find( word, letter):
        index = 0
        while index < len(word):
            if word[index] == letter:
                return index
            index = index + 1
        return -1

>>> find( 'Monty Python', 'P')
6

>>> find( 'Holy Grail', 'P')
-1
```

# Looping and counting

- A very common computer science pattern is the "counter"
  - search through a data structure
  - count the number of items found

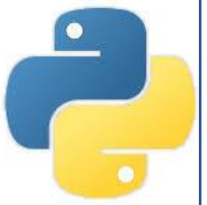
```
>>> word = 'banana'
>>> count = 0
>>> for letter in word:
        if letter == 'a'
            count = count + 1

>>> print(count)
3
>>>
```



# String methods

- the string is actually an "object" in Python
- strings have built-in support functions (technically called methods), e.g.
  - upper() makes it all UPPER CASE
  - lower() makes it all lower case
  - center( size, fillchar ) centers and fills
- you can access these functions using the dot operator ( "." )

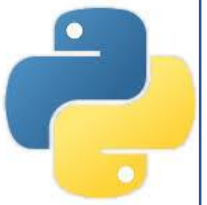
```
>>> word = 'banana'
>>> new_word = word.upper()
>>> new_word
'BANANA'
>>> word.center( 20, '*' )
'*******banana*******'
```



# The in operator

- The "in" operator is a Boolean used to test if one string is contained in another

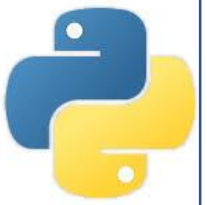
```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False

# try this - what happens?
>>> 'ana' in 'banana'

# find all the letters common to both strings
>>> def in_both( first, second ):
        for letter in first:
            if letter in second:
                print(letter)

>>> in_both( 'apples', 'oranges' )
a
e
s
```



# String comparisons

- Relational operators work on strings too
- To test for equality use ==
- Relational operators can be used for sorting in ascending and descending lexical order
- Note: upper case letters are considered "before" lower case letters

```
>>> word = 'banana'
>>> if word == 'banana':
        print( ' I found a banana')

# lexical ordering
>>> if word < 'banana':
        print(word + ' comes before banana')
elif word > 'banana':
        print(word + ' comes after banana')
else:
        print('your word is banana')
```



# Summary

- Python strings are arrays of characters
  - Strings are immutable - they can't be changed once created
  - New strings can be easily made from existing strings
- String object has a variety of useful features
  - len() for length, in tests for membership
- The slice operator stringvar[n:m] allows extraction of subsets of strings
  - starts at n, goes to m-1
  - missing n assumed to be 0
  - missing m assumed to be len(stringvar)-1