CSC 221: Introduction to Programming

Fall 2018

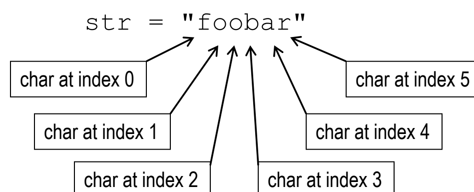
Text processing
- Python strings
- indexing & slicing, len function
- functions vs. methods
- string methods: capitalize, lower, upper, find, remove
- string traversal: in, indexing
- examples: palindromes, encryption

1

Python strings

so far, we have used Python strings to represent simple pieces of text
- displayed messages with a print statement
- passed in a name to a function
- the + operator concatenates strings, appends them end-to-end

strings are composite values, made of a sequence of individual characters
- can access individual characters using an index in brackets
- first character is at index 0; can specify negative index to count from end
- the built-in `len` function will return the length (number of chars) in a string




```
>>> str = "foobar"
>>> str[0]
'f'
>>> str[1]
'o'
>>> str[5]
'r'
>>> str[-1]
'r'
>>> len(str)
6
>>> str[len(str)-1]
'r'
```

2

Example: glitch

consider the following function

```
def glitch(word):
    return word[0] + "-" + word[0] + "-" + word[0] + "-" + word
```

what would be returned by `glitch("hello")` ?

note: you can multiply strings by an integer, appends that number of copies

```
def glitch(word):
    return (word[0]+"-")*3 + word
```

3

String slicing

can also slice off a substring by specifying two indices

`word[low:high]` evaluates to the substring, starting at index `low` and ending at `high-1`

if you omit either number, it assumes the appropriate end

`word[low:high:step]` can specify a step distance to skip characters

```
>>> word = "foobar"
>>> word[0:3]
'foo'
>>> word[3:6]
'bar'
>>> word[1:]
'oobar'
>>> word[::2]
'foa'
>>> word[::-1]
'raboof'
```

4

Example: rotating a string

consider the following function for rotating the characters in a string

```
def rotate_left(word):
    if word == "":
        return word
    else:
        first_char = word[0]
        rest_word = word[1:]
        return rest_word + first_char
```

- why do we need to test if `word == ""` ?

an error occurs if you index a character out of bounds
(but slicing ignores out-of-bounds indices)

```
>>> word = "foobar"
>>> word[6]
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    word[6]
IndexError: string index out of range
>>> word[1:100]
'oobar'
```

EXERCISE: define a `rotate_right` function that rotates the string in the opposite direction

5

Strings vs. primitives

like Turtles, Pixels and Images, Python strings are *objects*

- an *object* has its own *state* (here, a sequence of characters) and *methods* (functions that manipulate that state)
- *methods* are applied to a particular object and are dependent upon its internal state (e.g., the character sequence in a string object)

```
OBJECT.METHOD(PARAMETERS)
```

- e.g.,

```
>>> word = "foobar"
>>> word.capitalize()
'Foobar'
>>> word.upper()
'FOOBAR'
>>> word.lower()
'foobar'
>>> word.find("oo")
1
>>> word.find("z")
-1
>>> word.replace("oo", "u")
'fubar'
```

note: each of these methods returns a value based on the string's state – the string is NOT altered

6

Common string methods

capitalize( ) Return a copy of the string with only its first character capitalized
lower( ) Return a copy of the string converted to lowercase.
upper( ) Return a copy of the string converted to uppercase.

center(*width*) Return centered in a string of length *width.*
rjust(*width*) Return the string right justified in a string of length *width*.
strip( ) Return a copy of the string with the leading and trailing whitespace characters removed.

count(*sub*) Return the number of occurrences of substring *sub* in string. *Can provide two additional inputs, low & high, to limit to [low:high] slice.*
find(*sub*) Return the lowest index in the string where substring *sub* is found; -1 if not found. *Can similarly provide low & high inputs to limit the range.*
replace(*old, new*) Return a copy of the string with all occurrences of substring *old* replaced by *new*. *Can similarly provide low & high inputs to limit the range.*

7

Example: censoring words

what does the following function do?

```
def censor(word):
    word = word.replace("a", "*")
    word = word.replace("e", "*")
    word = word.replace("i", "*")
    word = word.replace("o", "*")
    word = word.replace("u", "*")
    return word
```

what if we wanted to censor capital vowels as well?

```
def censor(word):
    word = word.replace("a", "*")
    word = word.replace("e", "*")
    word = word.replace("i", "*")
    word = word.replace("o", "*")
    word = word.replace("u", "*")
    word = word.replace("A", "*")
    word = word.replace("E", "*")
    word = word.replace("I", "*")
    word = word.replace("O", "*")
    word = word.replace("U", "*")
    return word
```

this is getting tedious!

8

Looping through a string

behind the scenes of a for loop

- a for loop works because the built-in `range` function returns a sequence of numbers
  e.g., `range(5)` → `[0,1,2,3,4]`
- the variable `i` steps through each number in that sequence – one loop iteration per number
- if you don't need to access the varialbe inside the loop, can use _ instead

```
>>> for i in range(5):
        print(i)

0
1
2
3
4
```

```
>>> for _ in range(5):
        print("Howdy")

Howdy
Howdy
Howdy
Howdy
Howdy
```

similarly, can use a for loop to step through the characters in a string

```
>>> for ch in "foobar":
        print(ch)

f
o
o
b
a
r
```


Example: censor revisited

using a for loop, can greatly simplify our `censor` function

```
def censor(word):
    for ch in "aeiouAEIOU":
        word = word.replace(ch, "*")
    return word
```

EXERCISE: generalize the `censor` function so that the letters to be censored are provided as inputs

- note: lowercase and uppercase occurrences of the letters should be censored

```
>>> censor("aeiou", "foobar")
'f**b*r'
>>> censor("aeiou", "Oof dah Abba goo")
'**f d*h *bb* g**'
>>> censor("abcrst", "Frak")
'F**k'
```

Palindrome

suppose we want to define a method to test whether a word or phrase is a palindrome (i.e., same forwards and backwards, ignoring non-letters)

```
"bob"

"madam"

"Madam, I'm Adam."

"Able was I ere I saw Elba."

"A man, a plan, a canal: Panama."
```

if we ignore the non-letters issue, it's fairly straightforward

```
def reverse(word):
    return word[::-1]

def is_palindrome1(word):
    low_word = word.lower()
    return low_word == reverse(low_word)
```

11

Building up a string

to strip non-letters from a string, could try to

- call `remove` to remove every possible non-letter character
  way too many possibilities, most of which won't appear in the string

- traverse the string, character by character
- for each non-letter encountered, call `remove` to remove that letter
  could work, but inefficient (remove has to search for the char all over again)

better solution: build up a copy of the string, omitting non-letter characters

```
def word_copy(word):
    copy = ""
    for ch in word:
        copy += ch
    return copy
```

this simple function copies `word`, char-by char:

```
word= "foot"
copy = ""
     = "f"
     = "fo"
     = "foo"
     = "foot"
```

12

Example: strip_non_letters

to strip non-letters, make the character copying conditional

- traverse the string character by character, as in `word_copy`
- check each char to see if it is a letter (using the `isalpha` method)
- if it is, then concatenate it onto the copy; otherwise, ignore it

- can then use this in the final version of `is_palindrome`

```
def strip_non_letters(word):
    copy = ""
    for ch in word:
        if ch.isalpha():
            copy += ch
    return copy

def is_palindrome(word):
    low_word = word.lower()
    strip_word = strip_non_letters(low_word)
    return strip_word == reverse(strip_word)
```

13

Searching a string

suppose we want to determine if a character or substring appears somewhere in a string

- call `count` or `find` and test the result

```
>>> str = "banana"
>>> str.count("a") > 0
True
>>> str.count("z") > 0
False
```

```
>>> str.find("a") != -1
True
>>> str.find("z") != -1
False
```

- could also use `in` and `not in` to test this directly

```
>>> "a" in str
True
>>> "z" in str
False
>>> "a" not in str
False
>>> "z" not in str
True
```

```
>>> "ana" in str
True
>>> "baba" not in str
True
```

14

Exercise: generalize the stripping

recall that we generalized the `censor` function so that you could specify the characters to censor

```
def censor1(word):
    for ch in "aeiouAEIOU":
        word = word.replace(ch, "*")
    return word
```

```
def censor2(bad_chars, word):
    for ch in bad_chars:
        word = word.replace(ch, "*")
    return word
```

similarly, generalize the `strip_non_letters` function so that it has an additional input specifying the characters to strip

```
def strip_non_letters(word):
    copy = ""
    for ch in word:
        if ch.isalpha():
            copy += ch
    return copy
```

```
def strip_these(bad_chars, word):
    ???
```

15

Example: Caesar cipher

one of the earliest examples of encryption (secret codes for protecting messages) was the Caesar cipher

- used by Julius Caesar in 50-60 B.C. to encrypt military messages
- worked by shifting each letter three spots in the alphabet

  e.g., ET TU BRUTE → HW WX EUXWH

for each letter in the message:

- need to be able to find its position in the alphabet
- then find the character three spots later (wrapping around to the front for "XYZ")

- there are numerous ways of doing this
  simplest: construct a string made up of all the letters in the alphabet,
  use the `find` method to find the index of a char in that string,
  use indexing to find the char at (index+3)

16

Example: Caesar cipher

for simplicity, we'll assume the message is made of uppercase letters only

```
ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

def caesar(word):
    copy = ""
    for ch in word:
        index = ALPHABET.find(ch)
        next_index = (index + 3) % 26
        copy += ALPHABET[next_index]
    return copy
```

ALPHABET is *constant*, a variable that is assigned once and accessible to all

the convention is to write constant in all-caps

wrap-around is handled using the remainder operator

for the letter "Z", index = 25
nextIndex = (25+3)%26 = 28%26 = 2
so, "Z" → "C"

17

Exercise: Caesar update

enter the definition of the `caesar` function and try it out on different strings

```
ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

def caesar(word):
    copy = ""
    for ch in word:
        index = ALPHABET.find(ch)
        next_index = (index + 3) % 26
        copy += ALPHABET[next_index]
    return copy
```

what would happen if we changed the alphabet?

- e.g., the the Classical Roman alphabet did not have the letters 'J', 'U', or 'W'
- if we removed those letters from the ALPHABET string, would the function still work?

18

Avoiding dependencies in code

- the problem here is that calculation of `next_index` is dependent on the ALPHABET length (using the % operator)
  - if we change the ALPHABET string, we also have to change this number
- as a general rule in programming, we want to avoid these kinds of dependencies
  - when possible, write code so that a single change is propagated everywhere

```
ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

def caesar(word):
    copy = ""
    for ch in word:
        index = ALPHABET.find(ch)
        next_index = (index + 3) % 26
        copy += ALPHABET[next_index]
    return copy
```

```
def caesar(word):
    copy = ""
    for ch in word:
        index = ALPHABET.find(ch)
        next_index = (index + 3) % len(ALPHABET)
        copy += ALPHABET[next_index]
    return copy
```

19

Exercise: generalized rotation cipher

- generalize the Caesar cipher so that it can be used to perform any rotation
  - including negative rotations, which could be used to decode messages
  - rotate(3, str) → encode using Caesar cipher
  - rotate(-3, str) → decode using Caesar cipher
  - rotate(13, str) → encode/decode using rot13 (used in many online forums)

```
def caesar(word):
    copy = ""
    for ch in word:
        index = ALPHABET.find(ch)
        next_index = (index + 3) % len(ALPHABET)
        copy += ALPHABET[next_index]
    return copy
```

```
def rotate(num_chars, word):
    ???
```

20

Caesar revisited

once we have the generalized rotation cipher function

- we could use it to perform a Caesar cipher encodings/decodings

```
def rotate(num_chars, word):
    copy = ""
    for ch in word:
        index = ALPHABET.find(ch)
        next_index = (index + num_chars) % len(ALPHABET)
        copy += ALPHABET[next_index]
    return copy

def caesar_encode(word):
    return rotate(3, word)

def caesar_decode(word):
    return rotate(-3, word)
```

21