Strings

Kelly M. Thayer
Week 6 Lecture 5
2017-02-28

Contact info:
kthayer@wesleyan.edu
Office Hours: M 8:30-10:30 AM, T 10:10-11:15 AM, W 5:00-6:00 PM
Office: Exley 322/325
Announcements

• Upcoming Exam: **Tuesday March 7, 2016**
  • Administered during normal lecture time
  • No computers; paper based.
  • Please bring a pen or pencil
  • No cell phone use (including calculator)
  • Format: Multiple choice, short answer, long open ended problem solving
• If you require accommodations, please coordinate with your instructor
• This week’s HW on strings will be accepted until Thursday (instead of Tuesday) due to the exam
• Report to labs March 8 (Wed) & 9 (Thurs) as usual
Learning Objectives

• Recall information on strings from previous lectures
• Use built in string functions on strings
• Use the index operator on strings
• Use the slice operator to extract substrings
• Iterate over strings using while loops
• Compare strings
• Use the in operator to search for a substring within a string
• Write for loops to iterate over a string
• Combine new string skills with flow control and functions to write programs
Strings: A Quick Review
String Skills So Far

• Print

```python
>>> print ("Welcome to the Land O'Lakes!")
Welcome to the Land O'Lakes!
```

• Input

```python
>>> big_lake = input("Which of the five US Great Lakes it the largest?\n")
Which of the five US Great Lakes it the largest?
Superior
>>> big_lake
'Superior'
```

• Convert numeric values to strings

```python
>>> nlake = input('How many Great Lakes are fresh water lakes?\n')
How many Great Lakes are fresh water lakes?
5
>>> print('There are ', str(nlake), 'fresh water Great Lakes')
There are 5 fresh water Great Lakes
```

```python
>>> depth_km = input ("L. Michigan's max depth is 281m. How many km?\n")
L. Michigan's max depth is 281m. How many km?
0.281
>>> print("Lake Michigan's max depth is 281m, which is ", str(depth_km),"km.")
Lake Michigan's max depth is 281m, which is 0.281 km.
```
String Skills So Far, Cont'd

• Concatenate using +

    'Green Bay Packers'

    >>> team = 'Green ' + 'Bay ' + 'Packers'
    >>> team
    'Green Bay Packers'

• Repeat using *

    >>> cheer = (team + '!')*3
    >>> cheer
    'Green Bay Packers!Green Bay Packers!Green Bay Packers!'

• Compare for equality using ==

    >>> 'Green Bay Packers' == 'New England Patriots'
    False
Built ins Pertaining to Strings
Length Function

• The length function returns the number of characters in a string
• Spaces and punctuation do count as characters
• The length of the empty string is 0

Syntax:
len(<str>)

Examples:

>>> len('12345')
5
>>> len('1, 2, 3!') # 3 numeric + 3 punctuation + 2 spaces
8
>>> len('') # empty string
0
String Methods: Syntax

Syntax:
<variable>.<string_function>() # method notation

Example:
```python
>>> s = 'Huron'
>>> s.upper()
'HURON'
```

- s is the string 'Huron'
- `upper` is a function included in the `string` import.
- This function returns the input string s in upper case letters
String Methods:
Useful Methods Available

# isdigit  returns True if all characters of a nonempty string are numerical characters
```
>>> '123'.isdigit()
True
>>> 'abc'.isdigit()
False
```

# isalpha  returns True if all characters of a nonempty string are alphabetic characters
```
>>> '123'.isalpha()
False
>>> 'abc'.isalpha()
True
```

# islower  returns True if all alphabetic characters are lower case
```
>>> 'abc'.islower()
True
>>> 'AbC'.islower()
False
```

# issuper  returns True if all alphabetic characters are lower case
```
>>> 'ABC'.issuper()
True
>>> 'Abe'.issuper()
False
```
String Methods: More Useful Methods Available

# upper returns the alphabetic characters in the string in upper case

```python
>>> 'chicago'.upper()
'CHICAGO'
>>> 'Route66'.upper()
'ROUTE66'
```

# lower returns alphabetic characters in lower case

```python
>>> 'E E Cummings wrote 2900 poems'.lower()
'e e cummings wrote 2900 poems'
```
String Indexing
String Indexing

- The **index operator** looks up characters in a string
- The index operator is the square brackets `[ ]`
- An index is used to specify which character(s) in the string to extract
- Indices may be positive, negative, or zero
- A string can be considered by its characters and they can be indexed in the forward (positive) or backward (negative) orientation
- Indices > 0 begin at the left and tell the offset from that point
- Indices < 0 indicate offsets from the right end of the string
- Forward indexing begins with 0 and runs up to len-1

![Indexing Chart]

-8 -7 -6 -5 -4 -3 -2 -1

'M' 'I' 'C' 'H' 'I' 'G' 'A' 'N'

0 1 2 3 4 5 6 7
String Indexing: Syntax and Examples

• Syntax: `<string> [<index>]`

Examples:

```
>>> 'MICHIGAN' [0]  >>> 'MICHIGAN' [-8]
'M'
'PH'
>>> 'MICHIGAN' [1]  >>> 'MICHIGAN' [-7]
'I'
'I'
'C'
'C'
>>> 'MICHIGAN' [3]  >>> 'MICHIGAN' [-5]
'H'
'H'
'I'
'I'
>>> 'MICHIGAN' [5]  >>> 'MICHIGAN' [-3]
'G'
'G'
>>> 'MICHIGAN' [6]  >>> 'MICHIGAN' [-2]
'A'
'A'
>>> 'MICHIGAN' [7]  >>> 'MICHIGAN' [-1]
'N'
'N'
```

Each position within the string can be called in two ways, using either the forward or reverse system.
String Indexing: Exercise

Decode the answer to the question using the indicated indices.

\[ s_1 = 'SUPERIOR' \]
\[ s_2 = 'MICHIGAN' \]
\[ s_3 = 'ONTARIO' \]
\[ s_4 = 'HURON' \]
\[ s_5 = 'ERIE' \]
\[ s_6 = 'LAKE' \]

What animal was used to haul barges along the Erie Canal?
\[ s_2[0] \]
\[ s_4[1] \]
\[ s_6[-4] \]
\[ s_5[-1] \]
String Indexing: Exercise

Decode the answer to the question using the indicated indices.

\[
\begin{align*}
s_1 &= \text{'SUPERIOR'} \\
s_2 &= \text{'MICHIGAN'} \\
s_3 &= \text{'ONTARIO'} \\
s_4 &= \text{'HURON'} \\
s_5 &= \text{'ERIE'} \\
s_6 &= \text{'LAKE'}
\end{align*}
\]

The name of strings is often given as s as a convenient convention. Here we just number them.

What animal was used to haul barges along the Erie Canal? MULE

\[
\begin{align*}
>>> & s2[0] \\
& 'M' \\
>>> & s4[1] \\
& 'U' \\
>>> & s6[-4] \\
& 'L' \\
>>> & s5[-1] \\
& 'E'
\end{align*}
\]
Index Bounds

• Thus we see that in order to be a valid index for a string \( s \):

  • A non-negative integer \( i \) must satisfy
    \[ 0 \leq i \text{ and } i < \text{len}(s) \]

  ```python
  >>> s='ERIE'
  >>> s[0]
  'E'
  >>> s[3]
  'E'
  >>> s[4]
  Traceback (most recent call last):
    File "<pyshell#64>", line 1, in <module>
    s[4]
  IndexError: string index out of range
  ```

  • A negative integer \( j \) must satisfy
    \[ -\text{len}(s) < j \text{ and } j < 0 \]

  ```python
  >>> s2='HURON'
  >>> s2[-1]
  'N'
  >>> s2[-5]
  'H'
  >>> s2[-6]
  Traceback (most recent call last):
    File "<pyshell#69>", line 1, in <module>
    s2[-6]
  IndexError: string index out of range
  ```
Substrings: The Slice Operator
String Slicing

- Specifying a range to select more than one character is possible
- **slice operator** - returns the **substring** of a string between two indices
- The slice operator is `[ : ]`
- Use the colon to separate the start and stop indices
- The slice begins inclusive of the lower index
- The slice goes up to, but does not include, the upper index

**Syntax:**

```python
<string> [<start_index> : <stop_index>]
```

**Example:**

```python
>>> s = 'MICHIGAN'
>>> s[0:2]
'MI'
```

`s[0:2]` includes position 0 and goes up to but excludes position 2
String Slicing: Some Examples

```python
>>> s = 'MICHIGAN'
# s[0:len(s)] specifies the trivial case of the whole string

>>> s[0:len(s)]
'MICHIGAN'
```

```python
>>> s[0:8]
'MICHIGAN'
```

# if the start index is equal to the stop index, then the substring will be empty

```python
>>> s[2:2]
'
```

# if stop_index = start_index+1, then :
```
>>> s[start_index : stop_index] == s[start_index]
```

```python
>>> s[0:1]  # Slicing operator
'M'
```

```python
>>> s[0]  # Indexing operator
'M'
```
String Slicing: Some Examples

```python
>>> s = 'MICHIGAN'
```

# slicing beyond the length of the string adds nothing to the resulting string
```python
>>> s[0:100]
'MICHIGAN'
```

# if the start index is further right than the stop index, the empty string is returned
```python
>>> s[7:0]
''
```
String Slicing: Some More Examples

```python
>>> s = 'MICHIGAN'
# Negative indices can be used
>>> s[-7:-4]
'ICH'
```

# if the start index is further right than the stop index, the empty string is returned

```python
>>> s[-4:-7]
''
```

# positive and negative indices can be mixed

```python
>>> s[-5:5]
'HI'
```
String Slicing: Implied Indices

```python
g>>> s = 'MICHIGAN'
# If the start index is omitted, python begins at the beginning of the string

g>>> s[:4]
'MICH'
```

# If the stop index is omitted, python ends at the end of the string

```python
g>>> s[3:]
'HIGAN'
```

# If neither indices are specified, then the substring equals the string

```python
g>>> s[:]
'MICHIGAN'
```
Iteration over Strings
String Processing

- One can combine iteration with indexing to process a string
- Use a `while` loop

```python
# string processing

def loop_my_string(s):
    # signature: str -> NoneType
    # prints each character of the string

    i = 0 # initialization of the counter
    while i < len(s):
        print(s[i])
        i += 1 # augmented assignment
    return

>>> loop_my_string('abc')
a
b
c
```
String Processing
Spot the bugs!

```python
def count_s_bug(river):
    # signature: str -> int
    # counts times letter s appears
    # this version has bugs!
    i = 1
    counts = 0

    while i < len(river):
        if river[i] == 's':
            counts += 1

    return counts
```

• What will the output be? Test with ‘Mississippi’
• How can this be remedied?
• What can you do to help you debug the code?
String Processing: Working Example

```python
def count_s(river):
    # signature: str -> int
    # counts times letter s appears
    i = 0 # count for the loop
    counts = 0 # counter for number of s
    while i < len(river):
        if river[i] == 's':
            counts += 1
            i += 1
    return counts

>>> count_s("Mississippi")
4
```
String Comparisons
String Comparisons

- The **comparison operators** `<`, `<=`, `>`, `>=`, `==`, and `!=` are overloaded to compare strings
  - Within the same case, they are compared in alphabetical order
  - The set of upper case letters comes before the set of lower case letters

```plaintext
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

Examples:

# Being 'less than' means 'comes before'
```python
>>> 'a'<'b'  # 'a' comes before 'b'
True
```  

# Being 'greater than' means 'comes after'
```python
>>> 'Z'>'Y'  # 'Z' comes after 'Y'
True
```  

# Upper case letters come before lower case
```python
>>> 'Z'<'a'
True
```  

# Precedence moves from left to right as normal alphabetization
```python
>>> 'Z'<'a'
True
>>> 'Za'<'aZ'
True
>>> 'ZA'<'AZ'
False
>>> 'ZA'<'Az'
False
>>> 'za'<'az'
False
```
String Comparisons

0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

# Positive integers can be compared as strings, but...

```python
>>> '0' < '1'
True
```

# As strings precedence follows alphabetic convention and not numeric

```python
>>> 100 < 20
False
>>> '100' < '20'
True
```

# rules involving special characters such as punctuation, negative sign, decimal point, and so forth are ordered somewhat arbitrarily and need not be memorized

- If you need to know, look it up or test with Python
- Numbers should generally be compared using their proper type and not as strings
String Comparisons: T or F?

Exercise: Determine if python will return True or False:

• 'huron'.upper() > 'erie'.upper()

• 'SUPERIOR'.lower() < 'Michigan'

• 'OnTaRiO' < 'Ontario'

• 'OnTaRiO'.lower() != 'Ontario'.lower()
String Comparisons: T or F?

Exercise: Determine if python will return True or False:

• 'huron'.upper() > 'erie'.upper()
  >>> 'huron'.upper() > 'erie'.upper()
  True

• 'SUPERIOR'.lower() < 'Michigan'
  >>> 'SUPERIOR'.lower() < 'Michigan'
  False

• 'OnTaRiO' < 'Ontario'
  >>> 'OnTaRiO' < 'Ontario'
  True

• 'OnTaRiO'.lower() != 'Ontario'.lower()
  >>> 'OnTaRiO'.lower() != 'Ontario'.lower()
  False
in Operator
**in Operator**

- **in operator** – takes two strings as operands and returns a Boolean
  - Returns **True** if and only if the left operand can be found within the right operand
  - Otherwise it returns **False**

**Syntax:**
'<left_operand>' in '<right_operand>'

**Example:**
```
>>> 'Super' in 'Superior'
True
>>> 'Great' in 'Superior'
False
```

# two string operands

# Superior
in Operator: Examples

Examples:

# the substring must be consecutive

>>> 'Sir' in 'Superior'
False  # Superior returns False

# the search is case sensitive

>>> 'Ron' in 'Huron'
False  # Huron

# forward search only

>>> 'rat' in 'Ontario'
False  # Ontario
Write a function called `wordsearch` that:

• takes two strings `s1` and `s2` as input
• Checks if `s1` is a substring of `s2` and returns `True` if it is and `False` otherwise
• The search should be case independent
• Test with `on` in `Ontario`

```python
>>> wordsearch('on', 'Ontario')
True
```
Write a function called `wordsearch` that:
- Takes two strings `s1` and `s2` as parameters
- Checks if `s1` is a substring of `s2` and returns `True` if it is, and returns `False` otherwise
- The search should be case independent
- Test with `on` in `Ontario`

```
def wordsearch(s1, s2):
    # signature str, str -> Bool
    # case insensitive search for s1 in s2
    return s1.upper() in s2.upper()
```

```python
>>> wordsearch('on', 'Ontario')
True
```
For loops
for Loops with Strings

• Processing each character in a string in turn is a common task
• we already saw looping with while and a counter to do this earlier in the lecture
• A simplified looping construct exists: the **for loop**
• Automatically assign each character of the string to the loop variable in turn in the loop body; note there is no updated assignment explicitly; it’s automatic

Syntax:
```
for <loop_variable> in <string>:
    <loop_body>
```

Example:
```
for x in 'Erie':
    print(x)
```

# a new variable is defined
# a string
# may contain any number of lines of code
for Loop Example: Counting Vowels

- Write a function that will count how many vowels are present in some input string $s$
- Process $s$ using a `for` loop
- Call another function `is_vowel` to determine if each letter in $s$ is a vowel or not
  - Should be case independent
  - If it is, increment an accumulator `count` to keep track of how many vowels have been found as the loop progresses
  - increment the accumulator
  - Assume the vowels are `a`, `e`, `i`, `o`, `u`

```python
>>> count_vowels('Erie')
3
```
for Loop Example: Counting Vowels

def is_vowel(char):
    # signature: str -> bool
    # precondition: len(char)==1
    # tests if char is a vowel aeiou
    return char.lower() in 'aeiou'

def count_vowels(s):
    # signature: str -> int
    # returns number of vowels in a string
    count = 0
    for char in s:
        if is_vowel(char): # calls is_vowel
            count += 1
    return count

>>> count_vowels('Erie')
3
Loops: Exercise

Rewrite this while loop using a for loop instead.

```python
# while loop version

def count_s(river):
    # signature: str -> int
    # counts times letter s appears
    i = 0  # count for the loop
    counts = 0  # counter for number of s

    while i < len(river):
        if river[i] == 's':
            counts += 1
        i += 1
    return counts

>>> count_s('Mississippi')
4
```
Rewrite this while loop using a for loop instead.

```
# while loop version

def count_s(river):
    # signature: str -> int
    # counts times letter s appears
    i = 0 # count for the loop
    counts = 0 # counter for number of

    while i < len(river):
        if river[i] == 's':
            counts += 1
        i += 1

    return counts

# for loop version

def count_s_for(river):
    # signature: str -> int
    # counts times letter s appears
    # uses for loop instead of while
    counts = 0

    for i in river:
        if i == 's':
            counts += 1

    return counts
```

```
>>> count_s('Mississippi')
4

>>> count_s_for('Mississippi')
4
```
Strings Summary of Syntax

- `len(<string>)` # returns string length
- `<string>.isdigit()` # True if only contains digits
- `<string>.isalpha()` # True if only contains alpha
- `<string>.isupper()` # True if all upper case
- `<string>.islower()` # True if all lower case
- `<string>.upper()` # converts to upper case
- `<string>.lower()` # converts to lower case
- `<string>[<index>]` # returns char at position index
- `<string>[<start_index>:<stop_index>]` # slice
- `<string> <comparison_operator> <string>` # Returns Bool

- Looping over a string using while and counter
  
i = 0
  
  while i < len(s):
      <loop_body>
      i += 1

- `'<left_operand>' in '<right_operand>'` # Bool for substring
- Looping over a string using for loops
  
  for <loop_variable> in <string>:
      <loop_body>
THE END