Strings

CSE 1310 – Introduction to Computers and Programming
Alexandra Stefan
University of Texas at Arlington
Do you have the old book?

- *This chapter is the one with most differences between:*
  - the new and the old book,
  - 3.3 and 2.7 Python
  - The differences are in the string formatting that we show at the end of the slides.
Overview

• String objects:
  – How do you create a string object?
  – They are immutable.
• Element/character access
• Comparing strings: lexicographic order
• String operators: +, *, in
• Converting str objects to and from other types
• String methods
  – find(), index() ; strip(); split(); upper(); lower() ; isupper()...
• The ASCII code (UTF-8 for Python) : ord(), chr()
• Strings with special characters
• String formatting
String objects

- Words, sentences and whole phrases can be stored as string objects.

- Creating strings
  - Using single or double quotes:
    - 'These', " are ", ""string objects. ""
    - Double quotes are better than single: "Bob's" vs 'Bob's' vs 'Bob\'s'
    - The triple quotes preserve formatting (new lines). You can use them for longer comments.
  - Using the str constructor: str(123)

- String operators: +, *, in
  - Strings can be concatenated together with +
    >>> 'This' + ' is a ' + ' new string.'
  - String repetitions:
    >>> 'apple' * 3
    >>> 'apple' * 0
  - in: 'pl' in 'apple'
Converting Other Types to Strings

>>> a = 2012
>>> b = str(a)
>>> b
'2012'

• The `str` function converts objects of other types into strings.
Converting Strings into Ints/Floats

```python
>>> a = '2012'
>>> b = int(a)
>>> b
2012
>>> float(a)
2012.0
```

- The `int`, `float` functions convert strings to integers and floats.
  - Will give error message if the string does not represent an integer or float.

```python
>>> a = "57 bus"
>>> int(a)
<error message>
```
The **in** Operator

>>> vowels = "aeiou"
>>> "a" in vowels
True
>>> "k" in vowels
False

- **Syntax:**
  - **element in container**
- Returns **true** if the element appears in the container, **false** otherwise.
Strings: element access

- Individual characters (start counting from 0):
  ```
  >>> my_str = "Lovely"
  >>> my_str[0]
  >>> my_str[5]
  ```

- Slicing:
  ```
  >>> my_str[::2]  # a string formed from every other letter
  >>> my_str[st:ed:3]  # every 3rd letter starting from position st but stopping before ed
  >>> my_str[::-1]  # a copy of the string in reversed order
  ```

- `len` function gives the length of a string
  ```
  >>> len(my_str)
  ```
index and find

>>> my_str = "this is crazy"
>>> my_str.index("is")
2    # is this correct?
>>> my_str.index("q")
error

>>> my_str.find("is")
2
>>> my_str.find("q")
-1

• The `my_str.index(X)` method call returns the first position where X occurs in the string my_str.
  • Gives an error if X is not in my_str.

• The `my_str.find(X)` method call returns the first position where X occurs in the string my_str.
  • X is a string of one or more letters.
  • Returns -1 if X is not found.
  • Does not work for lists.
upper and lower

```python
>>> vowels = "aeiou"
>>> b = vowels.upper()
>>> vowels
'aeiou'
>>> b
'AEIOU'

>>> a = "New York City"
>>> b = a.lower()
>>> b
'new york city'
```

- The `string.upper()` method returns a new string where all letters are upper case.

- The `string.lower()` method returns a new string where all letters are lower case.

- Note: `upper()` and `lower()` do not modify the original string, they just create a new string.
  - Should be obvious, because strings cannot be modified.
The UTF-8 code (ASCII code)

• Characters are represented in the computer as numbers:
  – Python uses the UTF-8 encoding system
  – ASCII (the American Standard Code for Information Interchange) - 1963
    • earlier encoding system (128 characters), you may find references to it.
    • UTF-8 is backward compatible with it.
  – Unicode set – contains over a million characters from 93 scripts (alphabets) - 1980
  – UTF-8 (Universal Character Set Transformation Format – 8bit) – 1993

• Each letter is represented by a value or code (the ASCII value):
  – ord(ch) function returns the numeric value/code associated with the character given as an argument
    >>> ord('a')
    >>> ord('A')
    see others: '1', '.', '\n'
  – chr(i) function returns the character who’s value/code is i.
    >>> chr(97)

• ord() vs int()
  – Do not confuse the value/code associated with a character with the value of the character viewed as a number:
    >>> ord('1')
    >>> int('1')
### Subset of UTF-8

See Appendix D for the full set of ASCII

<table>
<thead>
<tr>
<th>Char</th>
<th>Dec</th>
<th>Char</th>
<th>Dec</th>
<th>Char</th>
<th>Dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>32</td>
<td>@</td>
<td>64</td>
<td>`</td>
<td>96</td>
</tr>
<tr>
<td>!</td>
<td>33</td>
<td>A</td>
<td>65</td>
<td>a</td>
<td>97</td>
</tr>
<tr>
<td>&quot;</td>
<td>34</td>
<td>B</td>
<td>66</td>
<td>b</td>
<td>98</td>
</tr>
<tr>
<td>#</td>
<td>35</td>
<td>C</td>
<td>67</td>
<td>c</td>
<td>99</td>
</tr>
<tr>
<td>$</td>
<td>36</td>
<td>D</td>
<td>68</td>
<td>d</td>
<td>100</td>
</tr>
<tr>
<td>%</td>
<td>37</td>
<td>E</td>
<td>69</td>
<td>e</td>
<td>101</td>
</tr>
<tr>
<td>&amp;</td>
<td>38</td>
<td>F</td>
<td>70</td>
<td>f</td>
<td>102</td>
</tr>
<tr>
<td>)</td>
<td>39</td>
<td>G</td>
<td>71</td>
<td>g</td>
<td>103</td>
</tr>
<tr>
<td>(</td>
<td>40</td>
<td>H</td>
<td>72</td>
<td>h</td>
<td>104</td>
</tr>
<tr>
<td>)</td>
<td>41</td>
<td>I</td>
<td>73</td>
<td>i</td>
<td>105</td>
</tr>
<tr>
<td>*</td>
<td>42</td>
<td>J</td>
<td>74</td>
<td>j</td>
<td>106</td>
</tr>
<tr>
<td>+</td>
<td>43</td>
<td>K</td>
<td>75</td>
<td>k</td>
<td>107</td>
</tr>
<tr>
<td>,</td>
<td>44</td>
<td>L</td>
<td>76</td>
<td>l</td>
<td>108</td>
</tr>
<tr>
<td>-</td>
<td>45</td>
<td>M</td>
<td>77</td>
<td>m</td>
<td>109</td>
</tr>
<tr>
<td>.</td>
<td>46</td>
<td>N</td>
<td>78</td>
<td>n</td>
<td>110</td>
</tr>
<tr>
<td>/</td>
<td>47</td>
<td>O</td>
<td>79</td>
<td>o</td>
<td>111</td>
</tr>
<tr>
<td>0</td>
<td>48</td>
<td>P</td>
<td>80</td>
<td>p</td>
<td>112</td>
</tr>
<tr>
<td>1</td>
<td>49</td>
<td>Q</td>
<td>81</td>
<td>q</td>
<td>113</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>R</td>
<td>82</td>
<td>r</td>
<td>114</td>
</tr>
<tr>
<td>3</td>
<td>51</td>
<td>S</td>
<td>83</td>
<td>s</td>
<td>115</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>T</td>
<td>84</td>
<td>t</td>
<td>116</td>
</tr>
</tbody>
</table>
String Comparisons

Given the strings:
"abc", "ABC", "Abc", "abcd", "acd", "a", "A", "c", "C"

Python would order them as follows:
'A', 'ABC', 'Abc', 'C', 'a', 'abc', 'abcd', 'acd', 'c'

• Python uses the **lexicographic (dictionary)** order for strings of different lengths
• Python uses the UTF-8 value to compare characters.
• Capital letters **are always before** lower case letters.
• Numbers **are always before** letters.
String Comparisons

• It is easy to verify the order that Python uses, by trying out different pairs of strings.

```python
>>> "hello" < "goodbye"

>>> "Hello" < "goodbye"

>>> "ab" > "abc"
```
String Comparisons

• It is easy to verify the order that Python uses, by trying out different pairs of strings.

```
>>> "hello" < "goodbye"
False
>>> "Hello" < "goodbye"
True

>>> "ab" > "abc"
False
```
String Comparisons

>>> "123" < "abc"

>>> "123" < "ABC"
String Comparisons

```python
>>> "123" < "abc"
True
>>> "123" < "ABC"
True
```

- Numbers come before letters.
- Comparing strings of different lengths.
  - Any prefix of a string is smaller than the string itself.
    ```python
    >>> "ab" > "abc"
    False
    ```
Strings are immutable

• Strings can not change:

```python
>>> a = "Munday"
>>> a[1] = 'o'
Traceback (most recent call last):
  File "<pyshell#297>", line 1, in <module>
a[1] = 'o'
TypeError: 'str' object does not support item assignment
```
If You Must Change a String...

• You cannot, but you can make your variable equal to another string that is what you want.

• Example:

```python
>>> my_string = "Munday"
   – my_string contains a value that we want to correct.

>>> my_string = "Monday"
   – We just assign to variable my_string a new string value, that is what we want.
```
For More Subtle String Changes...

• Suppose that we want a program that:
  – Gets a string from the user.
  – Replaces the third letter of that string with the letter A.
  – Prints out the modified string.
For More Subtle String Changes...

• Strategy:
  – convert the string to a list of characters
  – do any manipulations we want to the list (since lists can change)
  – convert the list of characters back to a string
    • Using a loop.
    • Using the `join` method for strings. – LATER(after lists)
      – `joining_string.join(list_of_strings)`
      – `>>> " - ".join([1,2,3])`
An Example

• Write a program that:
  – Gets a string from the user.
  – Modifies that string so that position 3 is an A.
  – Prints the modified string.
Solution with slicing and +

```python
my_string = input("please enter a string: ")

my_string = my_string[0:2] + "A" + my_string[3:]  
print("the modified string is ", my_string)
```
Escape sequences

• Escape sequences (non-printing characters)

  – these sequences will not be printed as they are, but produce a certain action (e.g. move the cursor to a new line):

  – \n  : new line (when printing, move to a new line)
  – \t  : tab (when printing, place a tab)
  – \\  : allows you to print the \ : 'This is a backslash: \ '\
  – \'  : allows you to print the ' : 'Bob\'s'
  – \  : the string continues on the next line (it will be printed on the same line)
  – \r  : moves the cursor at the beginning of current line (does not work in the Shell.)
Formatting operator: %

- Syntax of the formatting expression:
  
  ```
  format_string % (data1, data2,...)
  
  "_%__%__...._%" % (_____ , _____ ,...)
  ```

- Example:

  ```
  >>> "%s is %d years old" % ("Bill", 25)
  
  "%s is %d years old" % ("Bill", 25)
  ```

- Note: % also occurs in the format string.

- This expression evaluates to a string (it produces a string)

- Conversion specifier:

  ```
  %[flag][minimum_width] [.precision] [output type]
  
  "%s" % "Bill"  # see also my_str = "%s" % "Bill"
  "%10s" % "Bill"
  "%-10s" % "Bill"
  
  "%8.2f" % 2.3
  "%-8.2f" % 2.3
  "%8d" % 2.3
  ```
Conversion specifiers

%[flag][minimum_width] [.precision] [output type]

– Flags:
  • - : left aligned. Ex:
  • + : add sign symbol (+/-)
  • 0 : left pad with 0

– Output type: s (string);  d, i (integer);  f, g, e (float)

"%s" % "Bill"
"%10s" % "Bill"  # minimum width of 10
"%-10s" % "Bill"  # minimum width of 10 and left aligned

"%8.3f" % 2.3  # min width of 8 and 2 decimal points precision
"%-8.3f" % 2.3  # same as above and left aligned
"%08.3f" % 2.3  # padded with 0 to the left
"%8d" % 2.3    # output as an integer (no decimals)

# difference for using f, g, and e as output types
"%10.3f" % .0000023  # also try to format the display of .23
"%10.3g" % .0000023
"%10.3e" % .0000023
String methods

• You can find a list of the string methods in the Python Library Reference:
  ⇒ Go to: Sequence types – str, bytes,...
    ⇒ [link to Python Library Reference](http://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range)
    ⇒ scroll down to String Methods

– Sample methods: `split()`, `upper()`, `lower()`, `title()`, `isupper()`, `islower()`, `istitle()`
Predefined strings

```python
>>> import string  # this must be run before the following lines

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

>>> string.digits
'0123456789'

>>> string.whitespace
'\t\n\r\x0b\x0c'

>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'

>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```