CSc 110, Autumn 2017

Lecture 14: Strings

Adapted from slides by Marty Stepp and Stuart Reges
"Boolean Zen", part 1

• Students new to boolean often test if a result is True:
  ```python
  if is_prime(57) == True:  # bad
  ...
  ```

• But this is unnecessary and redundant. Preferred:
  ```python
  if is_prime(57):  # good
  ...
  ```

• A similar pattern can be used for a False test:
  ```python
  if is_prime(57) == False:  # bad
  if not is_prime(57):  # good
  ```
"Boolean Zen", part 2

• Functions that return bool often have an if/else that returns True or False:

    def both_odd(n1, n2):
        if n1 % 2 != 0 and n2 % 2 != 0:
            return True
        else:
            return False

• But the code above is unnecessarily verbose.
We could store the result of the logical test.

```python
def both_odd(n1, n2):
    test = (n1 % 2 != 0 and n2 % 2 != 0)
    if test:  # test == True
        return True
    else:     # test == False
        return False
```

Notice: Whatever `test` is, we want to return that.

- If `test` is True, we want to return True.
- If `test` is False, we want to return False.
Solution w/ "Boolean Zen"

• Observation: The if/else is unnecessary.
  • The variable test stores a bool value; its value is exactly what you want to return. So return that!

```python
def both_odd(n1, n2):
    test = (n1 % 2 != 0 and n2 % 2 != 0)
    return test
```

• An even shorter version:
  • We don't even need the variable test. We can just perform the test and return its result in one step.

```python
def both_odd(n1, n2):
    return (n1 % 2 != 0 and n2 % 2 != 0)
```
"Boolean Zen" template

• Replace

```python
def name(parameters):
    if test:
        return True
    else:
        return False
```

• with

```python
def name(parameters):
    return test
```
Improve the *is_prime* function

• How can we fix this code?

```python
def is_prime(n):
    factors = 0;
    for i in range(1, n + 1):
        if n % i == 0:
            factors += 1

    if factors != 2:
        return False
    else:
        return True
```
De Morgan's Law

- **De Morgan's Law**: Rules used to negate boolean tests.
  - Useful when you want the opposite of an existing test.

<table>
<thead>
<tr>
<th>Original Expression</th>
<th>Negated Expression</th>
<th>Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>a and b</td>
<td>not a or not b</td>
<td>not(a and b)</td>
</tr>
<tr>
<td>a or b</td>
<td>not a and not b</td>
<td>not(a or b)</td>
</tr>
</tbody>
</table>

- Example:

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Negated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if x == 7 and y &gt; 3:</td>
<td>if x != 7 or y &lt;= 3:</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Boolean practice questions

• Write a function named `is_vowel` that returns whether a `str` is a vowel (a, e, i, o, or u), case-insensitively.
  • `is_vowel("q")` returns False
  • `is_vowel("A")` returns True
  • `is_vowel("e")` returns True

• Change the above function into an `is_non_vowel` that returns whether a `str` is any character except a vowel.
  • `is_non_vowel("q")` returns True
  • `is_non_vowel("A")` returns False
  • `is_non_vowel("e")` returns False
Boolean practice answers

# Enlightened version. I have seen the true way (and false way)
def is_vowel(s):
    return s == 'a' or s == 'A' or s == 'e' or s == 'E' or s == 'i' or s == 'I'
    or s == 'o' or s == 'O' or s == 'u' or s == 'U'

# Enlightened "Boolean Zen" version
def is_non_vowel(s):
    return not(s == 'a') and not(s == 'A') and not(s == 'e') and not(s == 'E')
    and not(s == 'i') and not(s == 'I') and not(s == 'o') and
    not(s == 'O') and not(s == 'u') and not(s == 'U')

    # or, return not is_vowel(s)
Strings

• **string**: a type that stores a sequence of text characters.
  
  ```
  name = "text"
  name = expression
  ```

• Examples:
  
  ```
  name = "Daffy Duck"
  x = 3
  y = 5
  point = "(" + str(x) + ", " + str(y) + ")"
  ```
Indexes

• Characters of a string are numbered with 0-based *indexes*:

\[
\text{name} = "Ultimate"
\]

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-8</td>
<td>-7</td>
<td>-6</td>
<td>-5</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
</tr>
<tr>
<td>character</td>
<td>U</td>
<td>l</td>
<td>t</td>
<td>i</td>
<td>m</td>
<td>a</td>
<td>t</td>
<td>e</td>
</tr>
</tbody>
</table>

• First character's index : 0
• Last character's index : 1 less than the string's length
Accessing characters

• You can access a character with `string [index]`:
  ```python
  name = "Merlin"
  print(name[0])
  ```

  **Output:**  M
Accessing substrings

• Syntax:

```
part = string[start : stop]
```

• Example:

```
s = "Merlin"
mid = [1:3]     # er
```

• If you want to start at the beginning you can leave off start

```
mid = [:3]     # Mer
```

• If you want to start at the end you can leave off the stop

```
mid = [1:]     # erlin
```
# String methods

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>find(str)</code></td>
<td>index where the start of the given string appears in this string (-1 if not found)</td>
</tr>
<tr>
<td><code>substring(index1, index2)</code></td>
<td>the characters in this string from <code>index1</code> (inclusive) to <code>index2</code> (exclusive); if <code>index2</code> is omitted, grabs till end of string</td>
</tr>
<tr>
<td><code>substring(index1)</code></td>
<td></td>
</tr>
<tr>
<td><code>lower()</code></td>
<td>a new string with all lowercase letters</td>
</tr>
<tr>
<td><code>upper()</code></td>
<td>a new string with all uppercase letters</td>
</tr>
</tbody>
</table>

- These methods are called using the dot notation below:
  ```python
  starz = "Biles & Manuel"
  print(starz.lower())  # biles & manuel
  ```
String method examples

```python
# index     012345678901
s1 = "Allison Obourn"
s2 = "Merlin The Cat"

print(s1.find("o"))  # 5
print(s2.lower())    # "merlin the cat"
```

• Given the following string:

```python
# index     012345678901234567890123
book = "Building Python Programs"
```

• How would you extract the word "Python"?
Name border

- Prompt the user for full name

- Draw out the pattern to the left

- This should be resizable. Size 1 is shown and size 2 would have the first name twice followed by last name twice
Other String operations - length

• Syntax:

   length = len(string)

• Example:

   s = "Merlin"

   count = len(s)  # 6
Looping through a string

• The for loop through a string using range:

```python
major = "CSc"
for letter in range(0, len(major)):
    print(major[letter])
```

• You can also use a for loop to print or examine each character without range.

```python
major = "CSc"
for letter in major:
    print(letter)
```

Output:
C
S
C
# String tests

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>startswith(str)</code></td>
<td>whether one contains other's characters at start</td>
</tr>
<tr>
<td><code>endswith(str)</code></td>
<td>whether one contains other's characters at end</td>
</tr>
</tbody>
</table>

```python
def main():
    name = "Voldemort"
    if name.startswith("Vol"):
        print("He who must not be named")

    # Example: in keyword
    print("er" in name)  # True

if __name__ == '__main__':
    main()
```

- The **in** keyword can be used to test if a string contains another string.

  ```python
  example: "er" in name  # True
  ```
String question

• A Caesar cipher is a simple encryption where a message is encoded by shifting each letter by a given amount.
  • e.g. with a shift of 3, A → D, H → K, X → A, and Z → C

• Write a program that reads a message from the user and performs a Caesar cipher on its letters:

  Your secret message: Brad thinks Angelina is cute
  Your secret key: 3
  The encoded message: eudg wklqnv dqjholqd lv fxwh
Strings and ints

• All `char` values are assigned numbers internally by the computer, called ASCII values.

  • Examples:
    'A' is 65, 'B' is 66, ' ' is 32
    'a' is 97, 'b' is 98, '*' is 42

  • One character long strings and ints can be converted to each other
    `ord('a')` is 97, `chr(103)` is 'g'

• This is useful because you can do the following:
  `chr(ord('a' + 2))` is 'c'