CSC108H Lecture 3

Dan Zingaro
OISE/UT

May 28, 2009

Clickers: String Review (string_clicker.py)

```
def strange(s):
  return s.replace ('a', 'b').upper().upper()

print strange ('ab')
```

What is the output of this program?

- A. BA
- B. AB
- C. BB
- D. Error (because of the `.upper`)
- E. Error (because of the `.replace`)

Clickers: If Review (if_clicker.py)

```
val = 6
if val >= 4:
  print "one"
elif val > 2:
  print "two"
elif val == 6:
  print "three"
else:
  print "four"
```

What is the output of this program? (Output placed on one line)

- A. one two three
- B. three
- C. one
- D. one two three four
- E. two

Relational Operators

- Remember: = is the Python assignment operator
  - It is a command to evaluate the right-hand side and make the variable on the left refer to it
  - In math, = is a claim that two expressions are equal
- == is the Python operator that tests for equality
  - Other relational operators: `>` `>=` `<` `<=` `!=` (the last one being "not equal")
  - We typically use these in `if` statements

Exercise: Nested Ifs (if_ph.py)

Here, we have one `if` inside another (note the indentation!). Let's understand what's going on.

```
def ph_message():
  s = raw_input("Enter PH value: ")
  if len(s) > 0:
    ph = float(s)
    if ph < 7.0:
      print ph, "is acidic."
    elif ph > 7.0:
      print ph, "is basic."
    else:
      print ph, "is neutral."
  else:
    print "No pH value was given!"
```

Booleans

- New Python type: `bool`
- Contains only two values: `True` and `False`
- Relational operators like == and < give a `bool` back
  - e.g. `3 < 4` is `True`, and `4 < 3` is `False`
- We can assign the result of a comparison to a variable
  - e.g. `x = 3 < 4`
- . . . various `int` and `float` examples in shell

Logical Operators

- The logical operators take one (`not`) or two (`and`, `or`) bools and return a bool
- An expression involving `not` produces `True` if the original value is `False`, and `False` if the original value is `True`
- `And` produces `True` exactly when both of its operands are `True`
- `or` produces `True` exactly when at least one of its operands is `True`

```
snowing = False # examples
sunny = True
not snowing
not sunny
sunny and snowing
sunny or snowing
```

Logical Operators...

- In terms of precedence, `not` has the highest precedence, then `and`, then `or`

```
not a and b # with a and b False
(not a) and b
not (a and b)
a and b or True
a and (b or True)
```

What is True and False?

- We know that `True` is true and `False` is false
- The integer 0, the float 0.0, the empty string, the `None` object, and other empty data structures are considered `False`; everything else is `True`

```
if 0:
  print "hi"
if 4:
  print "bye"
```

Short-circuit Evaluation

- When an expression contains `and` or `or`, Python evaluates from left to right
- It stops evaluating once the truth value of the expression is known
- The value of the expression ends up being the last condition actually evaluated by Python

```
0 and 3 # examples
3 and 0
3 and 5
1 or 0
0 or 1
True or 1 / 0
...
```

String Comparisons

- We can use relational operators such as < and > on strings
- There is a well-defined ordering on strings: **a** comes before **b** which comes before **c** etc.
- This also holds for uppercase characters, and digits
- Python assumes that lowercase characters are > all uppercase characters which are > all digits
  - e.g. `'a' > 'C'`
- If one string is a prefix of another, it is considered <
- . . . various examples in shell

Clickers: Tricky Function (modfun_clicker.py)

```
def modfun(x):
  return x % 2 == 1
```

What does this function do?

- A. Nothing — the definition has an error
- B. Always returns 1
- C. Returns `True` exactly when the parameter is even
- D. Returns `True` exactly when the parameter is odd
- E. Returns `True` exactly when the parameter has the value 1

Looping through Strings

- we've seen "for" loops that loop through the pixels in a picture
- In Python, we can loop over anything that knows how to act as a sequence
- A string is a sequence of characters, much like a picture is sequentialized as pixels
- The pattern for looping through a string is as follows

```
for char in s:
  <do whatever, using char>
```

- The loop variable in the example above is `char`; it "steps through" each character in the string `s`

Looping through Strings...

- We will write several functions to process strings using a `for` loop
- Remember: Design-Code-Verify!
- Explicitly state what task you want to solve
- Refine the explanation into pseudo-code
- Verify the solution.

Example: How Many Vowels?

- Task: write a function that returns the number of vowels in a string
- Clarify: do we include "y"? Do we include lowercase and uppercase?
- Plan: loop through the string, testing each character against the vowels
  - Vowel? Yes — increment count
  - Vowel? No — do not increment count
- Let's code our solution ...

Possible Solution: How Many Vowels? (num_vowels.py)

```
def num_vowels(s):
  '''Return the number of vowels in string s.
  The letter "y" is not treated as a vowel.'''

  count = 0
  for char in s:
    if char in "aAeEiIoOuU":
      count += 1
  return count
```

Example: Removing Spaces

- Task: write a function that returns its input string with all spaces removed
- Can we remove from a string? What is the alternative?
- Plan: what should we do on each iteration?
- Let's code our solution ...

Possible Solution: Removing Spaces (remove_spaces.py)

```
def remove_spaces(s):
  '''Return s with any spaces removed.'''

  s_no_spaces = ""
  for char in s:
    if char != " ":
      s_no_spaces += char
  return s_no_spaces
```

Example: Number of Matches

- Task: write a function that takes parameters `s1` and `s2`, and returns the number of characters in `s1` that are also in `s2`
- Should a repeated character that is in `s1` count more than once?
- How do we test if a character is in a string?
- Plan: what should we do on each iteration?
- Let's code our solution ...

Possible Solution: Number of Matches (count_matches.py)

```
def count_matches(s1, s2):
  '''Return the number of chars in s1 that appear in s2'''

  common_chars = 0
  for char in s1:
    if char in s2:
      common_chars += 1
  return common_chars
```

More String Function Examples

- Write a function that reverses a string
- Write a function that returns `True` exactly when its two string inputs are equal (ignoring case of characters)
- Write a function that returns a string composed of the first, third, fifth, seventh ... characters from the input string

Nested Loops

- Just as we can nest `if` statements, we can nest other Python structures
- What does the following program involving nested loops do?

```
s = "abc"
t = "def"
for a in s:
  for b in t:
    print a, b
```

Strings and Indices

- Since a string is a sequence, we can use Python index notation to extract its characters
- Assume `s` is a string
- Then, `s[i]` for `i` ≥ 0 extracts character `i` from the left
- Be careful: the first character in a string has index 0, not 1!
- We can also use a negative index `i` to extract a character beginning from the right
- e.g. If `s = "abcde"`, then
  - `s[0]` is `a`
  - `s[1]` is `b`
  - `s[-1]` is `e`
  - `s[-3]` is `c`

Python Slice Syntax

- Python slice syntax allows us to extract a segment of characters from a string
- `s[i:j]` extracts characters beginning at `s[i]` and ending at the character one to the left of `s[j]`
- If we leave out the first index, Python defaults to using index 0 to begin the slice
- Similarly, if we leave out the second index, Python defaults to using index `len(s)` to end the slice
- `s[:]` therefore gives us a copy of the string
- We can use negative indices in the slice syntax as well

Python Slice Syntax...

- If `s = "abcde"`, then
  - `s[1:3]` is `bc`
  - `s[-2:-1]` is `d`
  - `s[-1:-2]` is the empty string
  - `s[:3]` is `abc`
  - `s[1:-3]` is `b`

Motivating While Loops: Another String Function (ask_and_match.py)

```
from count_matches import count_matches

def ask_and_match():
  '''Prompt the user to enter two strings,
  find out how many characters they have in common,
  and return the count as an int.'''

  word1 = raw_input("Enter a string: ")
  word2 = raw_input("Enter another string: ")
  common_count = count_matches(word1, word2)
  return common_count
```

Motivating While Loops

- Imagine we want this function to keep asking the user for two strings, until they have at least one character in common
- A similar scenario is asking the user for a command until a valid one is entered
- So far, we know about one type of loop: `for` loop
- Why can't we use a for-loop in this case? (Where's our sequence?)
- Another type of loop is the `while`: it repeatedly tests a condition, executing the body of the loop if it is True, and terminating the loop if it is False

String Function with While (ask_rep.py)

```
from count_matches import count_matches

def ask_rep():
  '''Prompt the user to enter two strings
  and find out how many characters they have in common.
  Repeat this task until the count is greater than 0,
  and return the count as an int.'''

  common_count = 0

  while common_count == 0:
    word1 = raw_input("Enter a string: ")
    word2 = raw_input("Enter another string: ")
    common_count = count_matches(word1, word2)

  return common_count
```