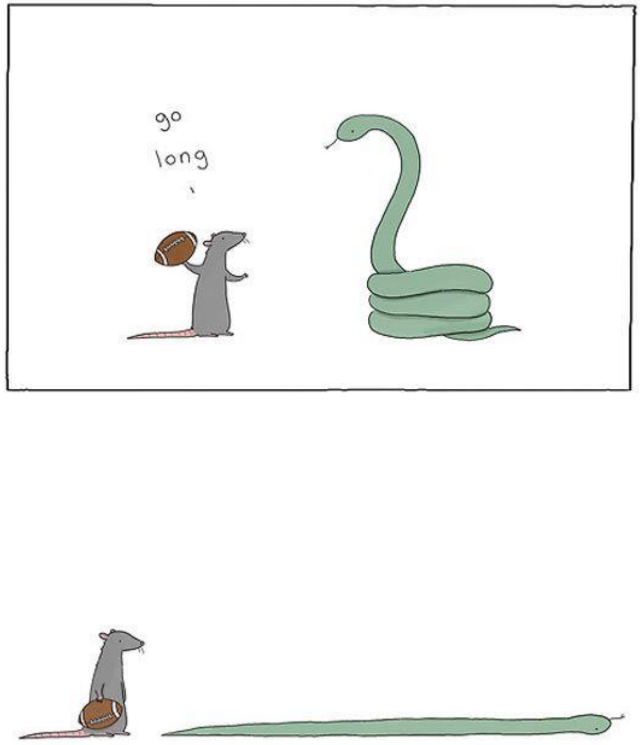
CSc 110, Spring 2017

Lecture 19: more with lists

Adapted from slides by Marty Stepp and Stuart Reges



lizclimo.tumblr.com

"Programs must be written for people to read, and only incidentally for machines to execute."

Abelson and Sussman,
Structure and Interpretation of Programs

Commenting Code

Comments are required for homework as follows:

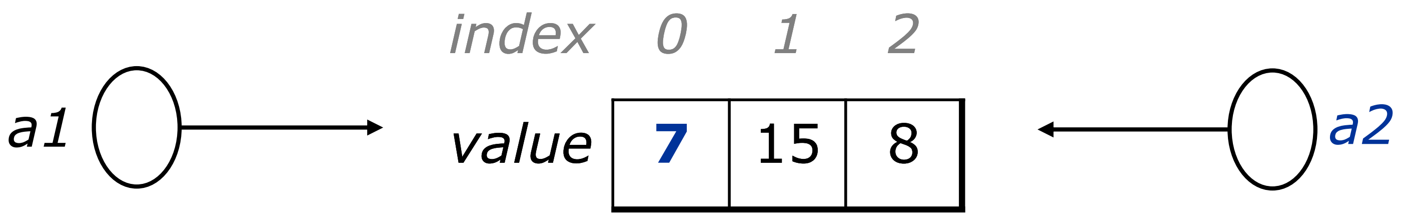
- at the top of the program file
- before each function
- within a function when needed to clarify a point (see below)

```
# Continue to loop until the user guesses the correct answer,
# giving a clue each time
while (guess != correct_answer):
    if (guess < correct_answer):
      print("It's higher.")
    else:
      print("It's lower.")
```

Lists and assignment

- Consider the following code:

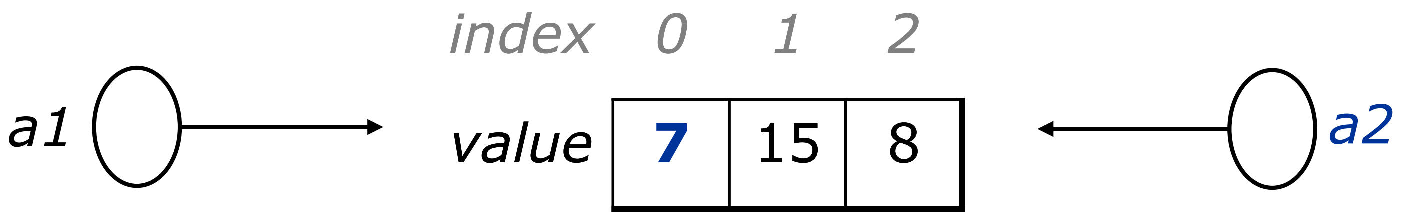
```
a1 = [4, 15, 8]
a2 = a1              # a2 now refers to same list as a1
a2[0] = 7
print(a1)            # [7, 15, 8]
```



Reference semantics

- When a type has *reference semantics*, a variable holds a reference to a value rather than the value itself. Lists have reference semantics.
  - Assigning a list to a variable causes the variable to hold a reference to the list
  - Modifying a list element referenced by one variable will affect any other variables referencing the same list.

```
a1 = [4, 15, 8]
a2 = a1             # a2 now refers to same list as a1
a2[0] = 7
print(a1)           # [7, 15, 8]
```



Consider the following interaction with Idle:

```
>>> a = [3, 7, 24]
>>> b = a
>>> print(b)
[3, 7, 24]

>>> b[0] = 88
>>> print(a)
[88, 7, 24]

>>> a[2] = 999
>>> print(b)
[88, 7, 999]

>>> a = [10, 20, 30]
>>> print(b)
[88, 7, 999]
```

Reference semantics
vs.
Value semantics

Value semantics

- When a type has *value semantics*, a variable holds a copy of a value.
  - ints, floats, strings and booleans in Python use value semantics.
  - When an int, float, string, or boolean value is assigned to a variable, its value is copied into memory set aside for the variable.
  - Assignment doesn't produces any sharing of data.
  - Modifying the value of one variable does not affect others.

```
x = 5
y = x
y = 17
x = 8
```

Integers as parameters

- Function square squares its parameter.

```
def square(x):
    x = x * x
```

- The value of variable a (of type int) is passed as an argument.

```
def main():
    a = 7

    # can variable a be modified?
    square(a)

    print(str(a))
```

The variable a cannot be modified by square.

Lists as parameters

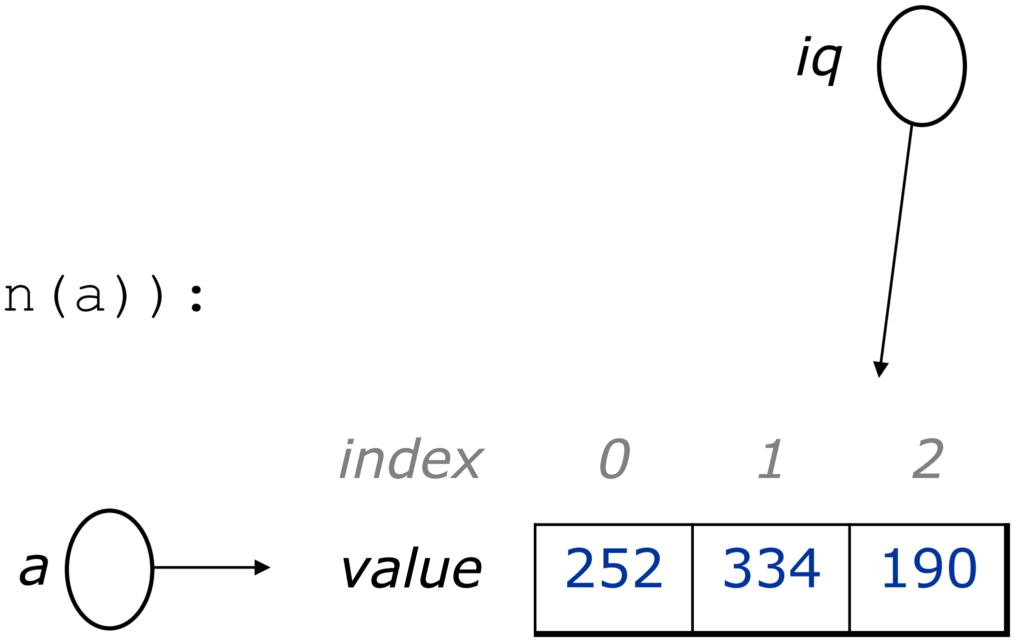
- Reference semantics apply not only to assignment but also to parameter passing.
    - Changes made in the function are also seen by the caller.

```
def main():
    iq = [126, 167, 95]
    double_all(iq)
    print(iq)

def double_all(a):
    for i in range(0, len(a)):
        a[i] = a[i] * 2
```

- Output:

```
[252, 334, 190]
```



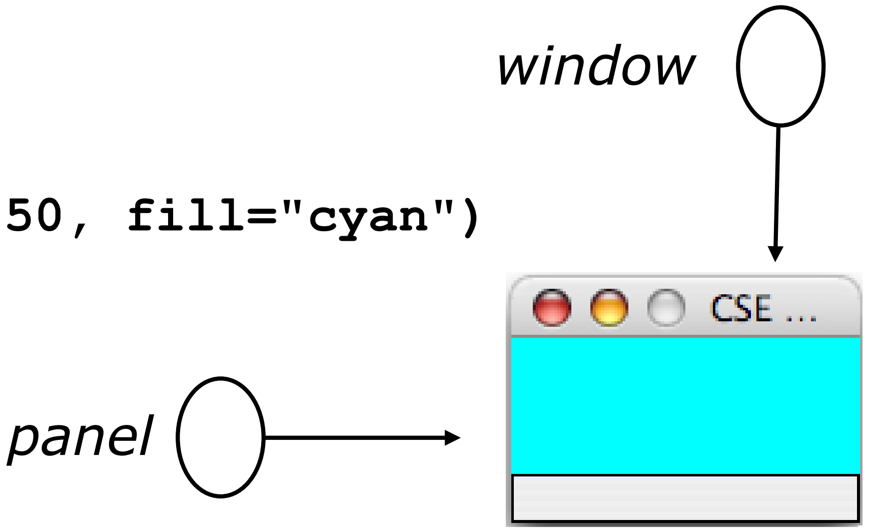
index	0	1	2
value	252	334	190

Objects as parameters

- Objects use reference semantics; the object is *not* copied. The parameter refers to the same object.
  - If the parameter is modified, it *will* affect the original object.

```
def main():
    window = DrawingPanel(80, 50)
    window.canvas.create_rectangle(0, 0, 80, 50, fill="yellow")
    example(window)

def example(panel):
    panel.canvas.create_rectangle(0, 0, 80, 50, fill="cyan")
 ...
```



Why reference semantics?

- Reference semantics.
  - *sharing.* It's useful to share an object's data among functions and methods.
  - *efficiency.* Copying large amounts of data can be inefficient.

```
f = open("population_data.txt")
data = f.readlines()    # data could be very large

process(data)           # a reference to data is passed in
  …
```

absolute_all

- Write a function absolute_all that accepts a list of integers and modifies it so that all its values are positive.

```
a = [-2, 15, 25, -106]
absolute_all(a)
print(a)           # [2, 15, 25, 106]
```

absolute_all

```
# Changes all values of the list to
# positive numbers.
def absolute_all(x):

    for i in range(0, len(x)):
        x[i] = abs(x[i])
```

rotate

- Write a function `rotate` that takes a list and rotates the first element to the end of the list

```
a = [10, 20, 30, 40]
rotate(a)
print(a)            # [20, 30, 40, 10]
```

- Hint: Use list's `append()` method.

rotate

```
# Rotates the list a by putting its first
# element at the end of the list.
def rotate(x):
    element = x.pop(0)
    x.append(element)
```

Problem: concat

- Write a function `concat` that accepts two lists and returns a new list containing all elements of the first list followed by all elements of the second.

- Note that this function *returns* a new list.

```
a1 = [12, 34, 56]
a2 = [7, 8, 9, 10]

a3 = concat(a1, a2)
print(a3)
# [12, 34, 56, 7, 8, 9, 10]
```

concat: v1

```
# Returns a new list containing all elements of x
# followed by all elements of y.
def concat(x, y):

    result = [0] * (len(x) + len(y))
    for i in range(0, len(x)):
        result[i] = x[i]
    for i in range(0, len(y)):
        result[len(x) + i] = y[i]
    return result
```

Question: Can we make this simpler?

concat: v2

```
# Returns a new list containing all elements of x
# followed by all elements of y.
def concat(x, y):

    result = []

    for item in x:
        result.append(item)
    for item in y:
        result.append(item)

    return result
```

Problem: concat3

- Write a function `concat3` that concatenates three lists similarly.

```
a1 = [12, 34, 56]
a2 = [7, 8, 9, 10]
a3 = [444, 222, -1]

print(concat3(a1, a2, a3))

# [12, 34, 56, 7, 8, 9, 10, 444, 222, -1]
```

concat3: v1 and v2

```
# Returns a new list containing all elements of x, y, and z.
def concat3(x, y, z):

    result = []
    for item in x:
        result.append(item)
    for item in y:
        result.append(item)
    for item in z:
        result.append(item)
    return result


# Shorter version that calls concat.
def concat3(a1, a2, a3):
    return concat(concat(a1, a2), a3)
```

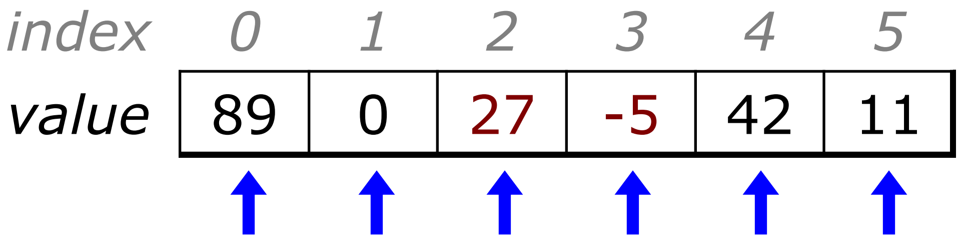
"When you hit a problem, you can lean forward and type or sit back and think."
-- Dr. Proebsting

List reversal question

- Write a function that reverses the elements of a list.
  - For example, if the list initially is this:
    `[11, 42, -5, 27, 0, 89]`
  - Then the list becomes:
    `[89, 0, 27, -5, 42, 11]`
    - Hint: think about swapping various elements...

Algorithm idea

- Swap pairs of elements from the edges; work inwards:



index	*0*	*1*	*2*	*3*	*4*	*5*
value	89	0	27	-5	42	11

Swapping values

```
a = 7
b = 35
# swap a with b?
a = b
b = a
print(str(a) + " " + str(b))
```

- What is wrong with this code? What is its output?

- The red code should be replaced with:

```
temp = a
a = b
b = temp
```

Flawed algorithm

- What's wrong with this code?

```
numbers = [11, 42, -5, 27, 0, 89]

# reverse the list
for i in range(0, len(numbers)):
    temp = numbers[i]
    numbers[i] = numbers[len(numbers) - 1 - i]
    numbers[len(numbers) - 1 - i] = temp
```

- The loop goes too far and un-reverses the array! Fixed version:

```
for i in range(0, len(numbers) // 2):
    temp = numbers[i]
    numbers[i] = numbers[len(numbers) - 1 - i]
    numbers[len(numbers) - 1 - i] = temp
```

reverse

- `reverse` – takes a list as a parameter and reverses it

```
numbers = [11, 42, -5, 27, 0, 89]
reverse(numbers)
```

- Solution:

```
def reverse(numbers):
    for i in range(0, len(numbers) // 2):
        temp = numbers[i]
        numbers[i] = numbers[len(numbers) - 1 - i]
        numbers[len(numbers) - 1 - i] = temp
```