CSCI 141

Lecture 23
Mutable objects and Functions
Goals

• Understand how mutable objects interact with function calls and scope:
  • Objects do not live inside the "boxes" that define scope
  • References to objects can cross "box" boundaries.

• Be able to draw memory diagrams for programs that involve function calls and mutable objects.
Implications of Mutability

• Previously: more than one variable (or list element) can contain *references* to the same object.

• Up next:
  • *Variables obey scope* (i.e., live in a certain "box").
  • *Objects don't*: they exist outside the "box" framework.
  • References can cross "box" boundaries.
Reminder:
How to Execute Function Calls

```python
def axpy(a, x, y):
    ''' Return a*x + y '''
    product = a * x
    result = product + y
    return result
```

```
> a1 = 2
> x1 = 3
> print(axpy(a1, x1, 4))
> print(a1)
```

Technically, global variables can be read but not modified from an inner scope.

In this class we'll always avoid referring to global variables from inside functions entirely.
Reminder: Function Calls

Execute this program:

```python
def xtty(x, y):
    """ return x ** y """
    return x ** y

a = 3
b = 2
print(xtty(a, b))
```
Reminder: Function Calls

Execute this program:

```python
def xtty(x, y):
    """ return x ** y """
    return x ** y

a = 3
b = 2
print(xtty(a, b))
```
Reminder: Function Calls

Execute this program:

```python
def xtty(x, y):
    """ return x ** y """
    return x ** y

a = 3
b = 2
print(xtty(a, b))
```
Reminder: Function Calls

Recall the steps to execute a function call:

1. Evaluate all arguments
2. Draw a local "box" inside the global one
3. Assign argument values to parameter variables in the local box
4. Execute the function body
5. When done, erase the local box
6. Replace the function call with its return value

```python
def xtty(x, y):
    """ return x ** y """
    return x ** y
```

```python
a = 3
b = 2
print(xtty(a, b))
```
Reminder: Function Calls

Recall the steps to execute a function call:

1. Evaluate all arguments

2. Draw a local "box" inside the global one

3. Assign argument values to parameter variables in the local box

4. Execute the function body

5. When done, erase the local box

6. Replace the function call with its return value

---

```python
def xtty(x, y):
    """ return x ** y """
    return x ** y
```

```python
a = 3
b = 2
print(xtty(a, b))
```
Reminder: Function Calls

Recall the steps to execute a function call:

1. Evaluate all arguments
2. Draw a local "box" inside the global one
3. Assign argument values to parameter variables in the local box
4. Execute the function body
5. When done, erase the local box
6. Replace the function call with its return value

```python
def xtty(x, y):
    """ return x ** y ""
    return x ** y
```

```python
a = 3
b = 2
print(xtty(a, b))
```
Reminder: Function Calls

Recall the steps to execute a function call:

1. Evaluate all arguments
2. Draw a local "box" inside the global one
3. Assign argument values to parameter variables in the local box
4. Execute the function body
5. When done, erase the local box
6. Replace the function call with its return value

```python
def xtty(x, y):
    """ return x ** y """
    return x ** y
```

```python
a = 3
b = 2
print(xtty(a, b))
```
Reminder: Function Calls

Recall the steps to execute a function call:

1. Evaluate all arguments
2. Draw a local "box" inside the global one
3. Assign argument values to parameter variables in the local box
4. Execute the function body
5. When done, erase the local box
6. Replace the function call with its return value

```python
def xtty(x, y):
    """ return x ** y """
    return x ** y
```

```python
a = 3
b = 2
print(xtty(a, b))
```

Recall the steps to execute a function call:
Reminder: Function Calls

Recall the steps to execute a function call:

1. Evaluate all arguments
2. Draw a local "box" inside the global one
3. Assign argument values to parameter variables in the local box
4. Execute the function body
5. When done, erase the local box
6. Replace the function call with its return value

```
def xtty(x, y):
    """ return x ** y """
    return x ** y
```

```
a = 3
b = 2
print(xtty(a, b))
```
Reminder: Function Calls

Recall the steps to execute a function call:

1. Evaluate all arguments
2. Draw a local "box" inside the global one
3. Assign argument values to parameter variables in the local box
4. Execute the function body
5. When done, erase the local box
6. Replace the function call with its return value

```python
def x tty(x, y):
    """ return x ** y """
    return x ** y
```

```python
a = 3
b = 2
print(x tty(a, b))
```
Mutable Objects and Functions
(or any mutable object!)

When you pass a list into a function, you're actually passing a *reference* to the list:

```python
def z1(a_list):
    a_list[0] = 0

a = [1, 1, 1]
z1(a)
print(a)
```

(1 1 1)
Mutable Objects and Functions

(or any mutable object!)

When you pass a list into a function, you're actually passing a reference to the list:

```python
def z1(a_list):
    a_list[0] = 0

a = [1, 1, 1]
z1(a)
print(a)
```

`a_list` points to the same object as the global variable `a`
Mutable Objects and Functions

(or any mutable object!)

When you pass a list into a function, you're actually passing a *reference* to the list:

```python
def z1(a_list):
    a_list[0] = 0

a = [1, 1, 1]
z1(a)
print(a)
```

![Diagram showing mutable object and function call]
Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```python
def z2(a_list):
    a_list = []

a = [1, 1, 1]
z2(a)
print(a)
```

```
  list
  0 1 2
  1 1 1
```

When you define a list (or any mutable object) inside a function, you're creating a new list that is separate from the global list. Therefore, modifying the list inside the function does not affect the global list:

- **a** is initially `[1, 1, 1]`
- After calling `z2(a)`, **a** becomes `[1, 1, 1]` again.
Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```python
def z2(a_list):
a_list = []

a = [1, 1, 1]
z2(a)
print(a)
```

The local variable `a_list` is reassigned to point to a **new** (different) list.

The list referenced by `a` is **unchanged**.
Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```python
def z2(a_list):
    a_list = []

a = [1, 1, 1]
z2(a)
print(a)
```

The local variable `a_list` is reassigned to point to a new (different) list.

The list referenced by `a` is unchanged.
Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```python
def z3(x):
    a_list = [x, x, x]
    return a_list

b = 2
a = z3(b)
print(a)
```

```
  def z3(x):
  a_list = [x, x, x]
  return a_list
  b = 2
  a = z3(b)
  print(a)

  int

  2

```
When you pass a list into a function, you're actually passing a reference to the list:

```python
def z3(x):
    a_list = [x, x, x]
    return a_list

b = 2
a = z3(b)
print(a)
```
Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```python
def z3(x):
    a_list = [x, x, x]
    return a_list

b = 2
a = z3(b)
print(a)
```

The function creates a new list, with the local variable `a_list` referring to it.
Mutable Objects and Functions

When you pass a list into a function, you're actually passing a reference to the list:

```python
def z3(x):
    a_list = [x, x, x]
    return a_list

b = 2
a = z3(b)
print(a)
```

The function creates a new list, with the local variable `a_list` referring to it.

A reference to the list is returned and assigned to `a`.
Again, with context

- **Variables obey scope** (i.e., live in a certain "box").
- **Objects don't**: they exist outside the "box" framework.
- References can cross "box" boundaries.

```python
def z3(x):
    a_list = [x, x, x]
    return a_list

b = 2
a = z3(b)
print(a)
```