

# Discussion 06

**Mutability, Iterators, Generators**

Aditya Balasubramanian

`aditbala [at] berkeley [dot] edu`

# Announcements

- Homework 4 is due Thursday 3/2.
- Lab policies

# Mutability



# List Mutation Functions

- `append(elem)`
  - box `elem` in list and add to end of list (can lead to nested lists)
  - only adds one element!
- `extend(elem)`
  - unbox `elem` and add to end (have to use an iterable)
- `insert(index, elem)`
  - insert `elem` at `index` (don't replace existing elem)
- `remove(elem)`
  - remove first appearance of `elem` in list (error if not found)
- `pop(index)`
  - removes and `return` elem at `index` (default arg is end of list)

# Mutating Lists

- List Mutation Functions modify **existing** list
- Slicing creates a **new** list
- `a = a + b` creates a **new** list
- `a += b` mutates **existing** list (basically `extend` )
- Indexing into list and changing values modifies **existing** list
  - `a = [1, 2, 3]`
  - `a[0] = 7`

# Identity vs Equality

- `is`
  - Check if two objects are the same (point to same reference in memory)
- `==`
  - Check to see if content is the same
- Demo

```
>>> a = [7, 6, 4]
>>> b = [7, 6, 4]
>>> a is b
False
>>> a == b
True
```

# Shallow Copy and Deep Copy

- Shallow Copy
  - What Python does most of the time
  - Copy top level of list
  - Point to same objects with nested list
- Deep Copy
  - Make completely new copy of list
  - Difficult to do this
- Whenever we copy a sequence, we are using a shallow copy

# Worksheet



# Iterators

- What are they?
  - An object that iterates over **iterables**, anything you can loop over
- Why use them?
  - Lazy evaluation
  - `hm`
  - `maybe this`

# Iterator Functions

- `next([iterator])`
  - get next element in iterator
  - When there are no more elements, return `StopIteration` Exception
- `iter([iterator])`
  - calling `iter` on an `[iterator]` will return the same iterator

# Iterator Example

```
>>> next(lst)                # Calling next on an iterable
TypeError: 'list' object is not an iterator
>>> list_iter = iter(lst)    # Creates an iterator for the list
>>> next(list_iter)          # Calling next on an iterator
1
>>> next(iter(list_iter))    # Calling iter on an iterator returns itself
2
>>> for e in list_iter:      # Exhausts remainder of list_iter
...     print(e)
3
>>> next(list_iter)          # No elements left!
StopIteration
>>> lst                      # Original iterable is unaffected
[1, 2, 3]
```

**WWPD**

# Generators

- What are they?
  - functions that return a custom iterator
  - When called, they return generator objects
- How do they work?
  - When you call `next` on the generator object, you evaluate all the code until the `yield` statement
  - At the `yield`, you stop and return to that line when `next` is called again

# Generator Function

```
def countdown(n):  
    print("Beginning countdown!")  
    while n >= 0:  
        yield n  
        n -= 1  
    print("Blastoff!")  
  
>>> c1, c2 = countdown(2), countdown(2)  
>>> c1 is iter(c1) # a generator is an iterator  
True  
>>> c1 is c2  
False  
>>> next(c1)  
Beginning countdown!  
2  
>>> next(c2)  
Beginning countdown!  
2
```

# yield from

- `yield from <iterable>`
  - `yield from [1,2,3,4]`
- `yield from <generator_func>`
  - recursive use of `yield from`
- equivalent to

```
for i in iterable:  
    yield i
```

```
>>> def rec_countdown(n):  
...     if n < 0:  
...         print("Blastoff!")  
...     else:  
...         yield n  
...         yield from rec_countdown(n-1)  
...  
>>> r = rec_countdown(2)  
>>> next(r)  
2  
>>> next(r)  
1  
>>> next(r)  
0  
>>> next(r)  
Blastoff!  
StopIteration
```



# Worksheet

# Thank you

Anon Feedback -> <https://tinyurl.com/adit-anon>