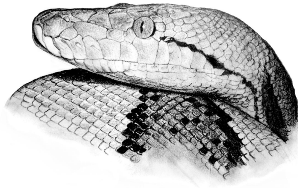
CS 61A Midterm 2 Review



Spring 2013

Outline

- Tuples, List, Dictionaries
- Recursion
- Nonlocal
- Environment Diagrams
- Equality vs. Identity
- Data Abstraction
- OOP
- Rlists

[1,2,3,4,5,6,7,8,9]
(1,2,3,4,5,6,7,8,9)
{1:2, 3:4, 5:6, 7:8}

Tuples, and Lists, and Dictionaries

(oh my!)

Tuples, Lists, Dictionaries

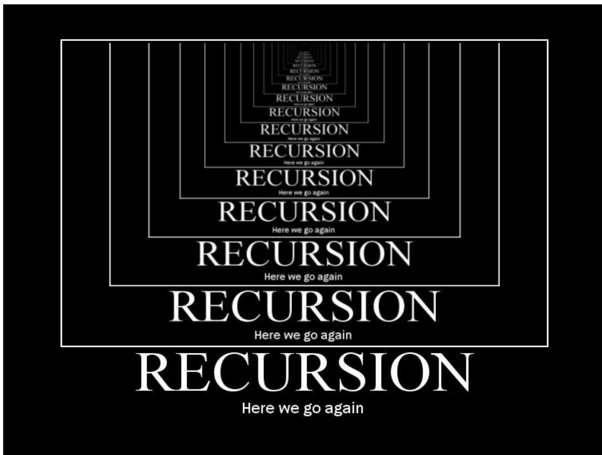
- Tuples
  - Hold elements in an *immutable* data structure
- Lists
  - Hold elements in a *mutable* data structure
- Dictionaries
  - Hold *key*, *value* pairs in a *mutable* data structure
    - Keys must be immutable

What does Python Display?

```
>>> a = (1,2,3,4)
>>> a[::-1]

__________
>>> a = a[:0:-1]
>>> a

__________
>>> b = [1,2,3,4]
>>> b[3] = a[1:]
>>> b

__________
>>> b[3][0] = a[:-2]
```

Write a function **`path_exists`** that takes in a dictionary, **`friends`** mapping every person to the list of their friends, and returns whether it is possible to move from the person **`start`** to the person **`finish`** by following friend relationships.

```
def find_path(friends, start, finish):
    """
    >>> allfriends = {"Soumya" : ["Julia"],      \
       "Julia": ["Mark", "Amir", "Soumya"],      \
       "Keegan" : ["Robert", "Sharad"]}
    >>> find_path(allfriends, "Soumya", "Mark")
    True
    >>> find_path(allfriends, "Soumya", "Keegan")
    False
    """
```



Recursion

- Divide a problem into smaller subproblems
  - It's like divide and conquer!
- Figure out the base case(s)
- When calling the recursive function, assume it works

Fibonacci

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

Mutating Map

Define a function m_map() that will recursively map a function to each element in a list, mutating the original one. It can be done in place.

```
def m_map(f, lst):
    """
    Takes in a list and recursively
    maps a function over each
    element, mutating the original.
    """
```

Deep Map

Write a function `deep_map(f, lst)` which applies a one-argument function onto every element in the given list. If an element is itself a list, then you should recursively apply the function onto each of its elements. You should NOT return anything—instead, mutate the original list (and any nested lists).

NonloCal

Nonlocal

- Tells Python that it is allowed to modify the binding for a declared variable in a parent frame
  - does not work for global variables
- Variable should already exist
- Python will *not* create a copy in the local frame

What does this function do?

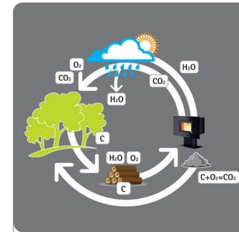
```
def make_mystery_sequence():
    n = 0
    x, y = 1, 1
    def mystery():
        nonlocal n, x, y
        if n == 0:
            n += 1
            return x
        elif n == 1:
            n += 1
            return y
        else:
            x, y = y, x + y
            return y
    return mytery
```

make_delayed_repeater()

Write a function that returns a function that returns the last thing it received (the first time it's called, it returns '...')

```
>>>slowpoke = make_delayed_repeater()
>>>slowpoke("hi")
...
>>>slowpoke("hello?")
hi
>>>slowpoke("stop repeating what I'm saying")
hello?
```

Environment Diagrams



Environment Diagram

```
def sillylist(mine, next):
    def cont():
        nonlocal cont
        cont = next
        return mine
    return cont

s = sillylist(1, sillylist(5, None))
s()
s()
```

Python Tutor

Another Environment Diagram

```
def go():
    def foo(a, b, c):
        return foo(a, b, c)
    def bar(a, b, c):
        return a+b+c
    def two():
        nonlocal foo, bar
        foo, bar = bar, foo
        return 2
    return foo(1, two(), 3)
print(go())
```

Python Tutor

Equality vs. Identity

Equality vs. Identity

- Equality
  - checks if two items are equivalent
  - use the == operator
    - internally, Python calls `__eq__()`
- Identity
  - checks if two items are the same object
    - stronger condition than equality
  - use the `is` operator
    - internally, Python calls `__is__()`

Equality vs. Identity

```
>>> l1, l2 = list(range(5)), list(range(5))
>>> l1 == l2
______
>>> l1 is l2
______
>>> l2 = l1
>>> l1 is l2
______
>>> d1, d2 = {1: 3, 5: 7}, {5: 7, 1: 3}
>>> d1 == d2
______
>>> d1 is d2
______
```

Data Abstraction

Data Abstraction

- We want to store data, i.e. numbers, strings, etc. in an organized way that allows us (and others!) to use it easily.

- Two major concerns:
  - How we store the data (lists, tuples, other data structures)
  - How we use the data (constructors, selectors)

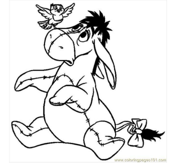
How do we represent data types?

- In Python, we have several ways
  - Object oriented programming
  - Data Abstraction with constructors and selectors
  - Dispatch Functions
  - ...

Data Abstraction

```
def make_rlist(first, rest):
   return (first, rest)

def first(rlist):
   return rlist[0]
def rest(rlist):
   return rlist[1]

#make sure you don't violate abstraction!
def popped1(rlist):
   return rlist[1]
def popped2(rlist):
   return rest(rlist)
```

Dispatch Functions



```
def make_donkey(name):
   weight_carried = 0
   max_carry = 5
   def dispatch(msg):
      if msg=='carry':
         nonlocal weight_carried
         weight_carried+=1
      if msg=='talk':
         if dispatch("alive"):
            print("hee-haw, my name is", name)
      if msg=='alive':
         #Dies if you made it carry too much
         return weight_carried<=max_carry
   return dispatch
```

Object Oriented Programming

The Donkey from Earlier as a Class

```
class Donkey:
   max_carry = 5
   def __init__(self,name):
      self.weight_carried = 0
      self.name = name
   def carry(self):
         self.weight_carried += 1
   def talk(self):
      if self.alive:
         print("hee-haw, my name is",self.name)
   @property
   def alive(self):
      return self.weight_carried <= max_carry
```

Create a new `AdultDonkey` class that Implements Breeding

```
>>> d = AdultDonkey("Dopey")
>>> e = AdultDonkey("Jazz")
>>> f = d.breed_with(e)
>>> f.talk()
hee-haw, my name is Dopey Jr.
>>> for _ in range(10):
...     f.carry()
>>> d.talk()
hee-haw, my name is Dopey
>>> f.talk()
```

Name credit: http://www.wookeyfarm.com/2012/05/99-donkey-names/

Let's also make a `StrongDonkey` that can carry up to 100 things inclusive

```
>>> s = StrongDonkey("arnold schwarzedonkey")
>>> for _ in range(100):
...   s.carry()
>>> s.alive
True
>>> s.carry()
>>> s.alive
False
>>> s.breed
<bound method ...>
```

Mutable Rlist Class

Mutable Rlist Class

```
class Rlist(object):
    class EmptyList(object):
        def __len__(self):
            return 0
    empty = EmptyList()

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __len__(self):
        return 1 + len(self.rest)

    def __getitem__(self, i):
        if i == 0:
            return self.first
        return self.rest[i - 1]
```

insert()

Write a function `insert` that takes in an `Rlist`, an index, and a value, inserting the value at the index position in the `Rlist`. It should mutate the original `Rlist`.

```
def insert(rlist, index, value):
    """Mutatively insert VALUE at INDEX in the
    RLIST."""
```

sort_rlist(rlist)

Write a function to sort a given rlist in an increasing order. You may assume that values stored in the rlist are integers.

>>>sort_rlist(Rlist(1, Rlist (3, Rlist (2,None))))
Rlist(1, Rlist(2, (Rlist (3, None))))