Day 5: Data, Functions, & Classes

Suggested reading: *Learning Python* (3rd Ed.)

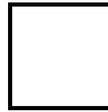
Chapter 15: Function Basics
Chapter 16: Scopes and Arguments
Chapter 22: *OOP: The Big Picture [optional]*
Chapter 23: Class Coding Basics [skim]
Chapter 24: Class Coding Details [skim]

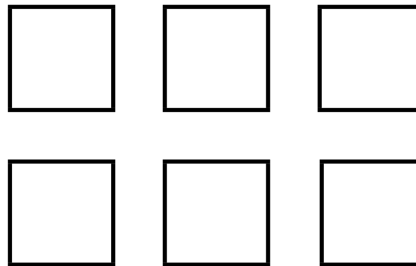
Turn In Homework

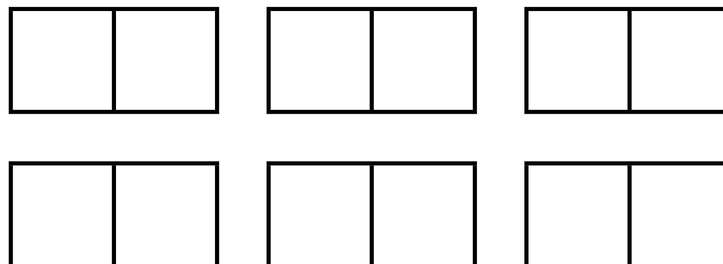
Homework Review

Data Structures

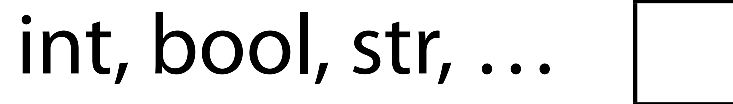
Data Structure Review

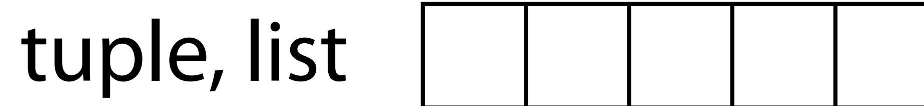


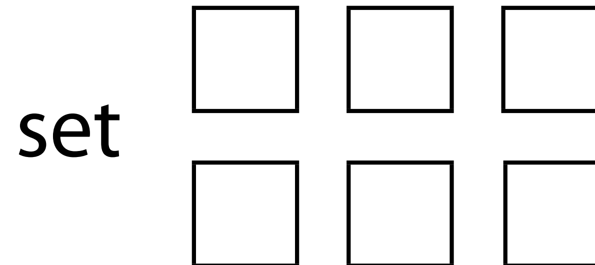


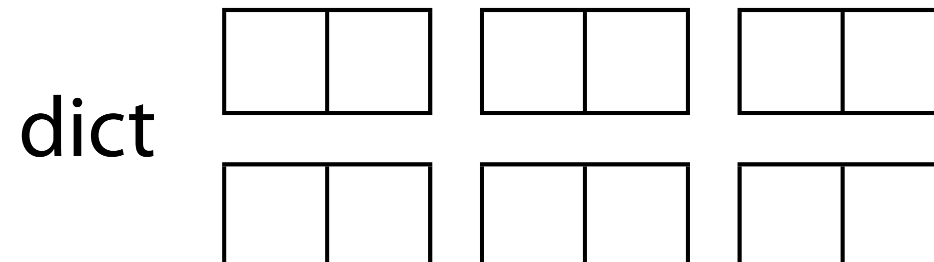


Data Structure Review









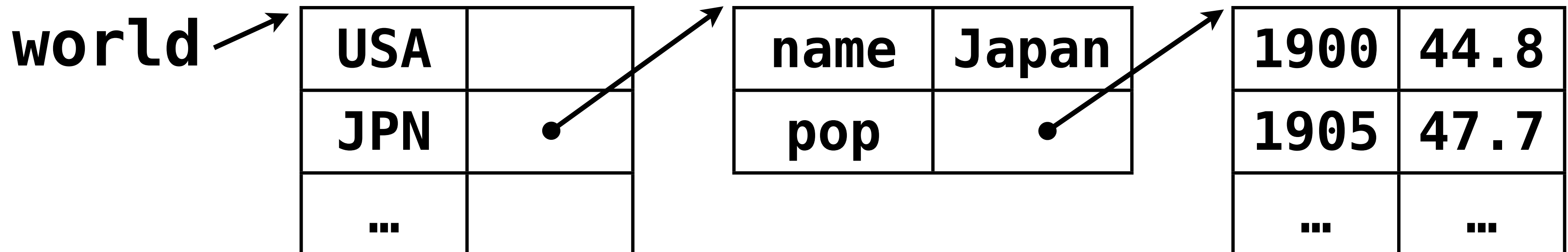
Complex Data Structure Examples

- Complex mappings
  - Country code => country info, yearly statistics
  - User => Service => set of IP addresses
  - Experimental condition (*N* vars) => *M* measures
- Multidimensional array (aka, a matrix):
  - Markov chain of *N* matrices, each X × Y
  - Coordinate transformations
  - Other stuff typically done in MATLAB…
- Trees and graphs
  - Genealogical tree
  - Network topology with latency measurements

Nested Data Structures

- Trivial in Python: Nest objects within collections
- Single value can be a tuple, list, set, dict
- Dictionary keys must be immutable; can use tuples

```
world = {'JPN': {'name': 'Japan',
                 'pop': {1900, 44.8}},
         'USA': … }
world['USA']['pop'][1900] = 76.2
```



Creating a Complex Structure

```
# world['USA']['pop'][1900] = 76.2

world = {}

world['USA'] = {'name': 'United States',
                'pop': {}}
world['USA']['pop'][1900] = 76.2
world['USA']['pop'][1901] = ...

world['JPN'] = {}
world['JPN']['name'] = 'Japan'
world['JPN']['pop'] = {}
world['JPN']['pop'][1900] = ...
```

Functions

Why Use Functions?

- Maximize code reuse /
  Minimize code redundancy
- Organize code clearly (decomposition)
- Make testable units of code
- Like a script within a script

Creating a Function

```
def function():
    <statement 1>
    <more statements>
```

- Creates **function** object
- Assigns object to function name
- ***Does not execute statements!***

```
def greet_world():
    print 'Hello, world!'
    print '2 + 2 =', str(2 + 2)
    print 'And now, goodbye.'
```

Using a Function

```
function()
```

- Actually runs code

```
def greet_world():
    print 'Hello, world!'
    print '2 + 2 =', str(2 + 2)
    print 'And now, goodbye.'

greet_world()
print '-' * 20
greet_world()
```

Function Arguments

```
def function(argument1, argument2, ...):
    # Can use argument variables here

function(42, 'Tim')
```

- Provides input to a function — if needed!
- Argument variables initialized by *assignment* (**=**)
- Thus, think about **`y = x`** and mutable/immutable

```
def greet_person(name):
    print 'Hello, %s!' % (str(name))
greet_person('Tim')
greet_person(raw_input('Enter name: '))
```

Default and Named Arguments

```
def foo(a, b, c=None, d=42):
    print a, b, c, d
```

```
foo(1, 2)                       => 1, 2, None, 42
foo(1, 2, 3)                    => 1, 2, 3, 42
foo(1, 2, 3, 4)                 => 1, 2, 3, 4

foo(b=6, a=89)                  => 89, 6, None, 42
foo(4, 3, d=12)                 => 4, 3, None, 12
foo(d=1, a=2, b=3, c=4)         => 2, 3, 4, 1
```

- Default arguments are useful and common
- Named arguments can be useful, less common

Function Return Values

```
def function(...):
    # Do stuff
    return some_value
```

- Identifies the output of the function
- Returns any single object (not named variable)
- Can occur more than once, anywhere in function

```
def f2c(f):
    if type(f) != float: return None
    return (f - 32.0) * 5 / 9

c = f2c(57.5)
```

Variable Scoping: Assignment

```
y = 0
def linear_1(x):
    # ...
    y = 2 * x + 1
    print 'Inside:', y
linear_1(42)
print 'Outside:', y
```

- Separate contexts to search for variable name:
  - Local scope is within one function *call*
  - Global scope is in same file (module), but not in **def**
- Local ***assignment*** hides global name
- Override local scope with **global** declaration

Variable Scoping: Assignment

```
y = 0
def linear_2(x):
    global y
    y = 2 * x + 1
    print 'Inside:', y
linear_2(42)
print 'Outside:', y
```

- Separate contexts to search for variable name:
  - Local scope is within one function *call*
  - Global scope is in same file (module), but not in **`def`**
- Local ***assignment*** hides global name
- Override local scope with **`global`** declaration

Variable Scoping: No Assignment

```
a = 3
b = 7
def linear_3(x):
    y = a * x + b
    return y

print linear_3(42)
```

- If *only* referencing a variable, search (in order):
  - Local scope
  - Global (module) scope
  - Built-in scope (cannot change)
- Otherwise, raise an exception

Variable Scoping: No Assignment

```
a = 3
def linear_4(x):
    b = 7
    y = a * x + b
    return y

print linear_4(42)
```

- If *only* referencing a variable, search (in order):
  - Local scope
  - Global (module) scope
  - Built-in scope (cannot change)
- Otherwise, raise an exception

Classes and Objects

What *Are* Objects and Classes?

- **Object**
  - Collection of related data
  - Actual memory with **value(s)**
  - Has a **type**, which is its class…
- **Class**
  - Definition of a kind of object
  - Encapsulates data ***and*** code
  - Pattern for building an object
  - Contains the **functions** that work on the data

box
height length width
set_size(h, l, w) volume() can_hold(h, l, w)

Defining a Class: Code

```
class class_name(object):
    def function1(self): <...>
    def function2(self, ...): <...>
```

- These functions will work on objects of this class
- First argument is `self`

```
class box(object):

    def volume(self):
        return ...

    def holds(self, height, len, width):
        return ...
```

Defining a Class: Object Data

```
class class_name(object):

    def __init__(self):
        self.var1 = ...

    def function1(self, ...):
        print self.var1
```

- Object data is created by assignment
- No explicit declaration
- Define data in `__init__()`, called for new object
- Use data in any function with `self.` prefix

Class Definition Example

```
class box(object):

  def __init__(self, height, length, width):
    self.height = height
    self.length = length
    self.width = width

  def volume(self):
    return self.height * self.length * self.width

  def can_hold(self, height, length, width):
    return (height <= self.height) and \
           (length <= self.length) and \
           (width <= self.width)

...
```

Using a Class

```
class class_name(…): …
x = class_name(…)
x.variable = 42
x.function(…)
```

```
s = ' Hello '        # or str(' Hello ')
print s.strip()

l = []               # or list()
l.append('a')

b = box(5, 7, 2)
if b.can_hold(3, 2, 1):
    print 'can hold volume:', b.volume()
```

Last 2 Slides!

Other Scripting Languages

- **Data structures**
  - Easy in some (e.g., Ruby, JavaScript)
  - Harder in others (e.g., Perl)
- **Functions** — YES! everywhere — but different:
  - Syntax
  - Argument options
  - Scope rules
- **Classes:** only in some (e.g., Ruby, sort of JavaScript)

Homework

- Read and store world country & population data
- Report on population of a country & its % of whole
- BE SURE TO LABEL YOUR PRINTOUT!!!

```
#!/usr/bin/env python

"""Homework for CS 368-4 (2011 Fall)
Assigned on Day 05, 2011-11-08
Written by <Your Name>
"""
```