Day 5: Data, Functions, & Classes

Suggested reading: *Learning Python* (4th Ed.)

Chapter 16: Function Basics
Chapter 17: Scopes
Chapter 18: Arguments

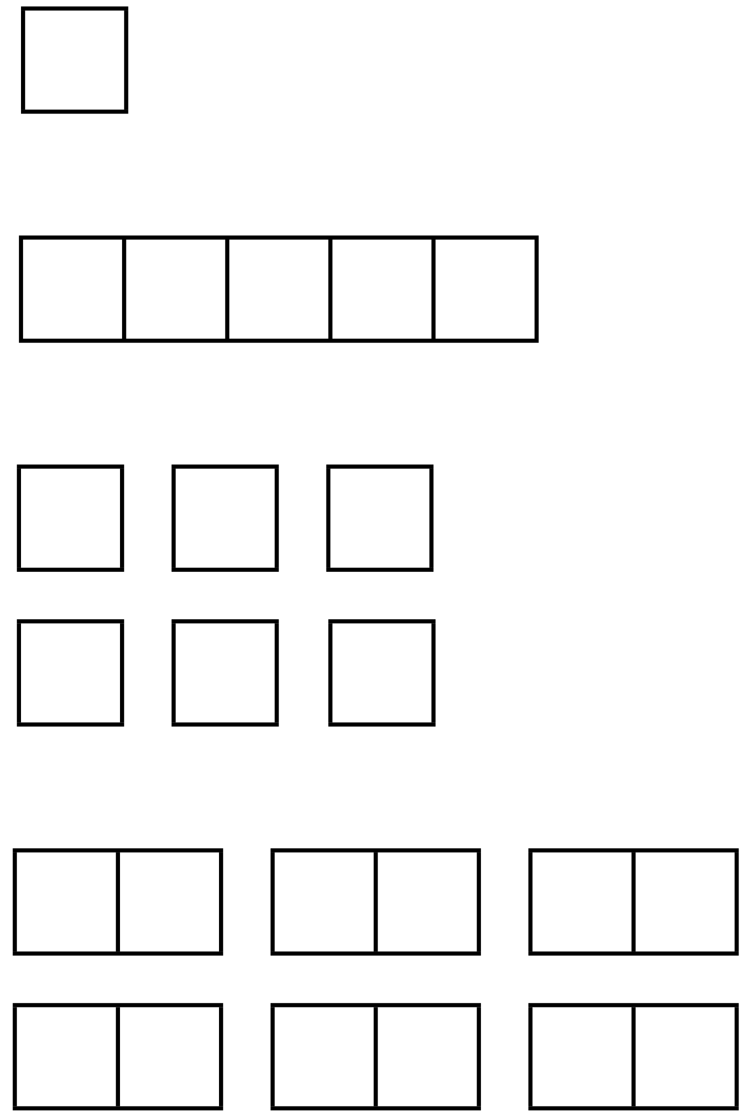
Chapter 25: *OOP: The Big Picture [optional]*

Turn In Homework

Homework Review

Data Structures

Data Structure Review



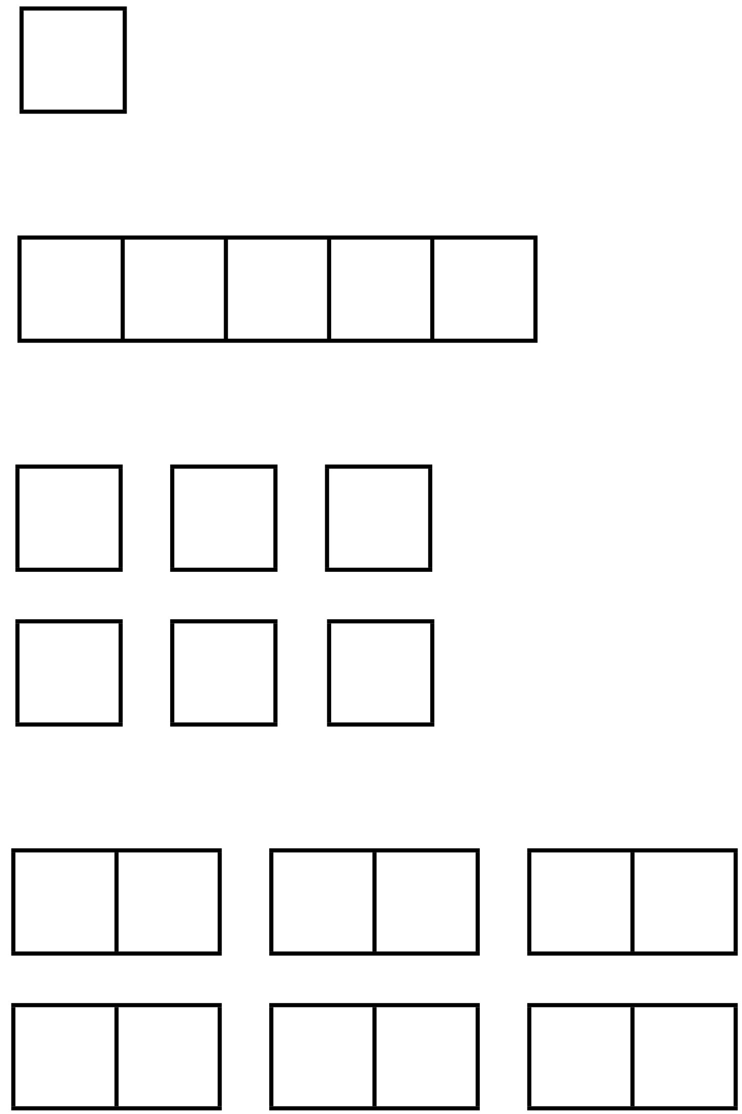
Data Structure Review

int, bool, str, …

tuple, list

set

dict

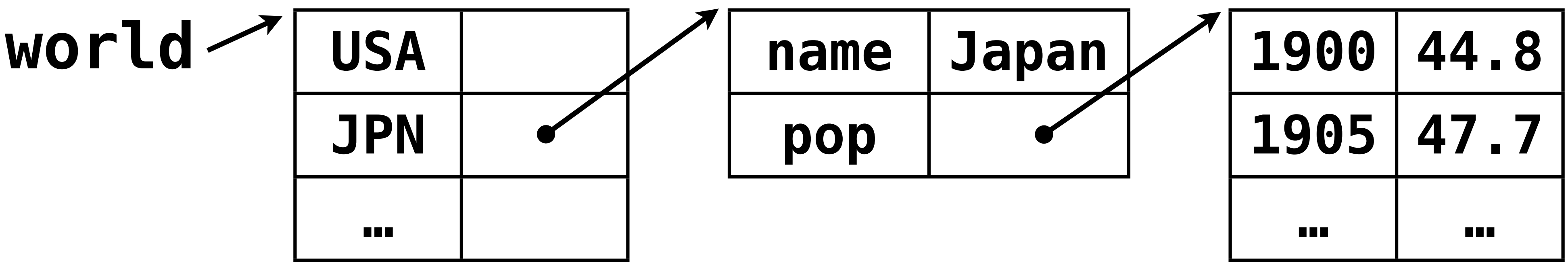


Complex Data Structure Examples

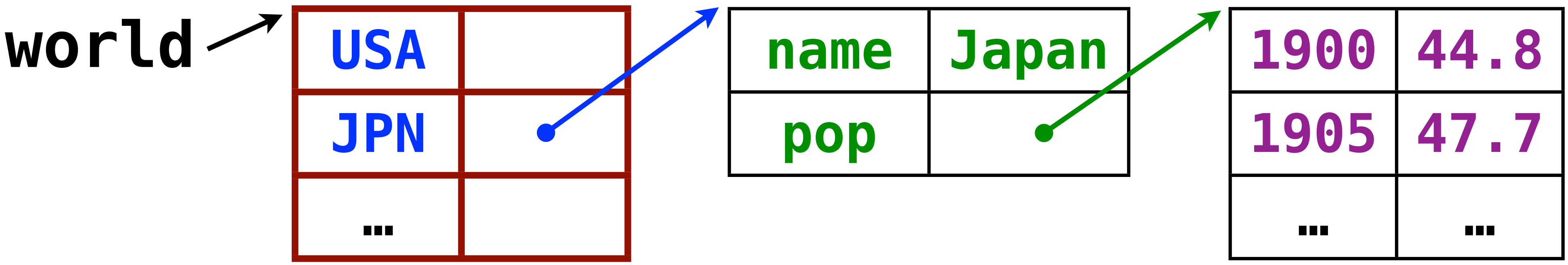
- Complex mappings
  - Country code => country info, yearly statistics
  - User => Service => set of IP addresses
  - Experimental condition (*N* vars) => *M* measures
- Multidimensional array (aka, a matrix):
  - Markov chain of *N* matrices, each X × Y
  - Coordinate transformations
  - Other stuff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

Nested Data Structures

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

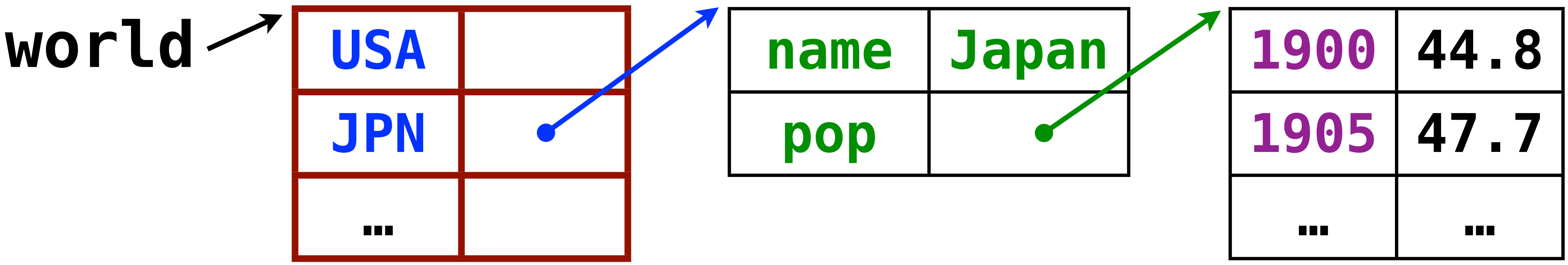


Creating a Complex Structure I



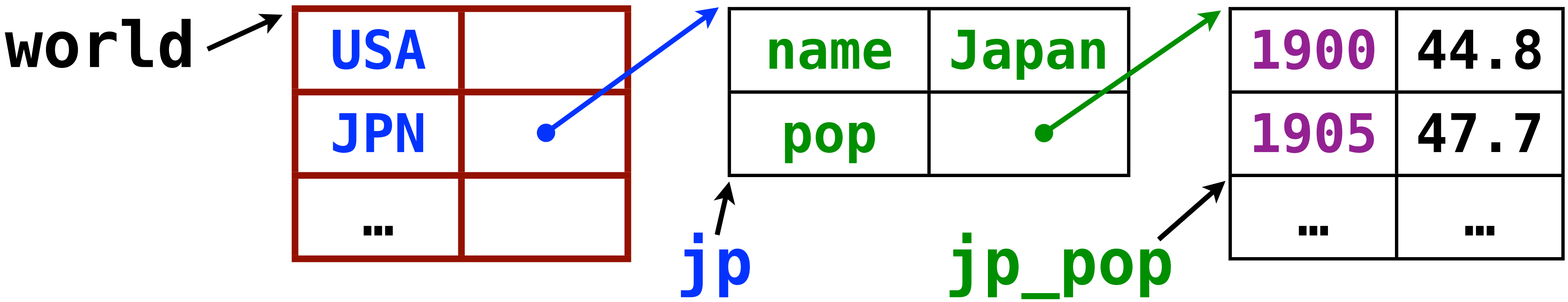

```
world = {
          'USA': {
            'name': 'United States',
            'pop': {
              1900: 76.2,
              1901: ...
                    }
                  }, ...
          'JPN': {...}, ... }
```

Creating a Complex Structure II



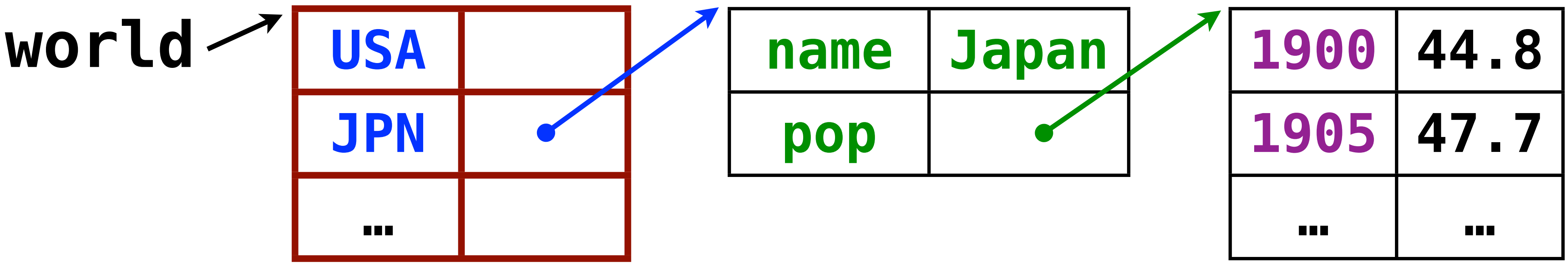

```
world = {}

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

world['JPN'] = {'name': 'Japan', 'pop':{}}
```

Creating a Complex Structure III




```
jp_pop = {1900: 44.8, 1905: 47.7, ...}
jp = {'name': 'Japan', 'pop': jp_pop}

us_pop = {1900: 76.2, ...}
us = {'name': 'United States',
      'pop': us_pop}

...

world = {'JPN': jp, 'USA': us, ...}
```

Using a Complex Structure



- Essentially: Just chain the “lookups” in each part
- Think hard about expressions and values
- Use **print** and **type()** to debug!

```
print type('USA')                 # <type 'str'>
print type(world['USA'])          # <type 'dict'>
print world['JPN']['name']              # 'Japan'
print world['JPN']['pop'][1900]         # 44.8
print 'Name: %s' % (world['JPN']['name'])
```

Thought (or Code) Experiments

- Review examples on Slide 6
  - Diagram data structure
  - Sketch code for creation and use
- Ideas from your own work?
- These experiments are not part of your homework (which uses the country data), but if you are stuck or have questions, I would be happy to try to help!

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()
```

Functions Are …

Functions Are …

- Like (almost) everything else in Python…

Functions Are …

- Like (almost) everything else in Python…
- ***OBJECTS!***

Functions Are …

- Like (almost) everything else in Python…
- ***OBJECTS!***
- As always, we must ask: Immutable or mutable?

Functions Are …

- Like (almost) everything else in Python…
- ***OBJECTS!***
- As always, we must ask: Immutable or mutable?

```
def hello():
    print 'Hello from hello()'
print hello     # <function hello at 0x...>

goodbye = hello
def hello():
    print 'And now for something ...'

hello()
goodbye()
```

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

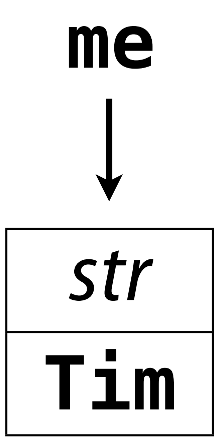
```
def add_person(name, alist):
    name = "'%s'" % (name)
    alist.append(name)

add_person('Tim', instructors)
```

Arguments Are *Assigned*

```
def addp(name, alist):
    name = 'Dr. ' + name
    alist.append(name)

me = 'Tim'
all = []
addp('Tim', all)
print me, all
```

Arguments Are *Assigned*

```
def addp(name, alist):
    name = 'Dr. ' + name
    alist.append(name)

me = 'Tim'
all = []
addp('Tim', all)
print me, all
```

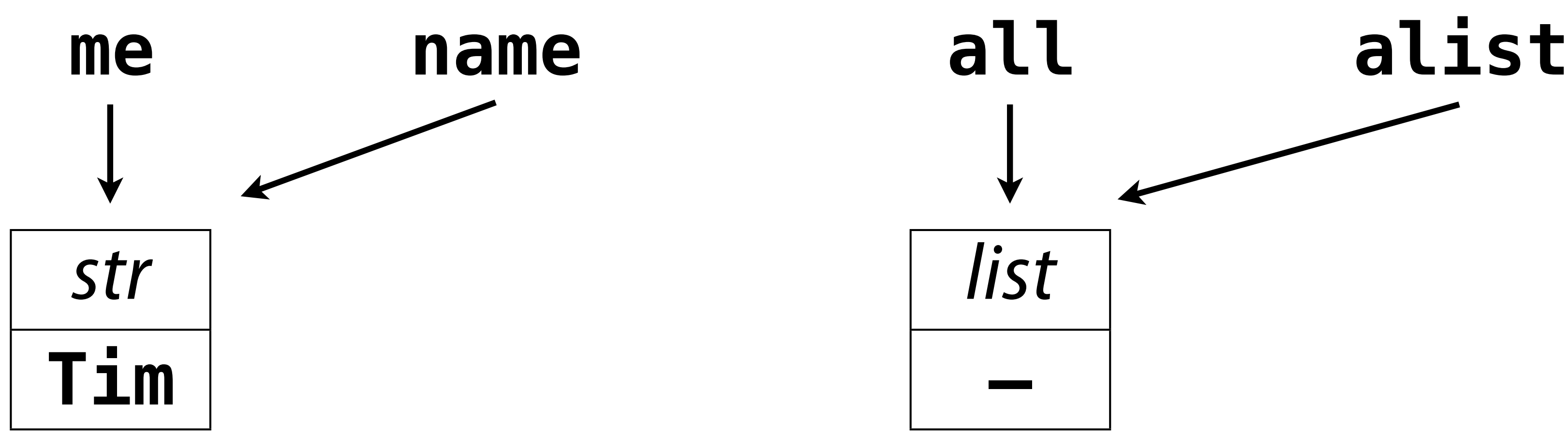


Arguments Are *Assigned*

```
def addp(name, alist):
    name = 'Dr. ' + name
    alist.append(name)

me = 'Tim'
all = []
addp('Tim', all)
print me, all
```

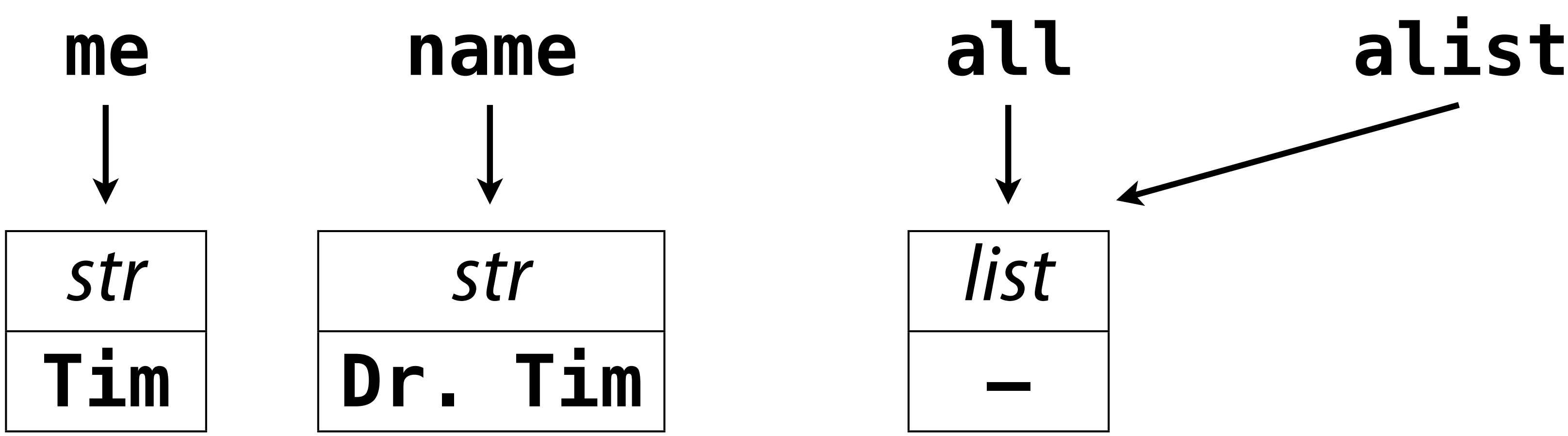
Arguments Are *Assigned*

```
def addp(name, alist):
    name = 'Dr. ' + name
    alist.append(name)

me = 'Tim'
all = []
addp('Tim', all)
print me, all
```



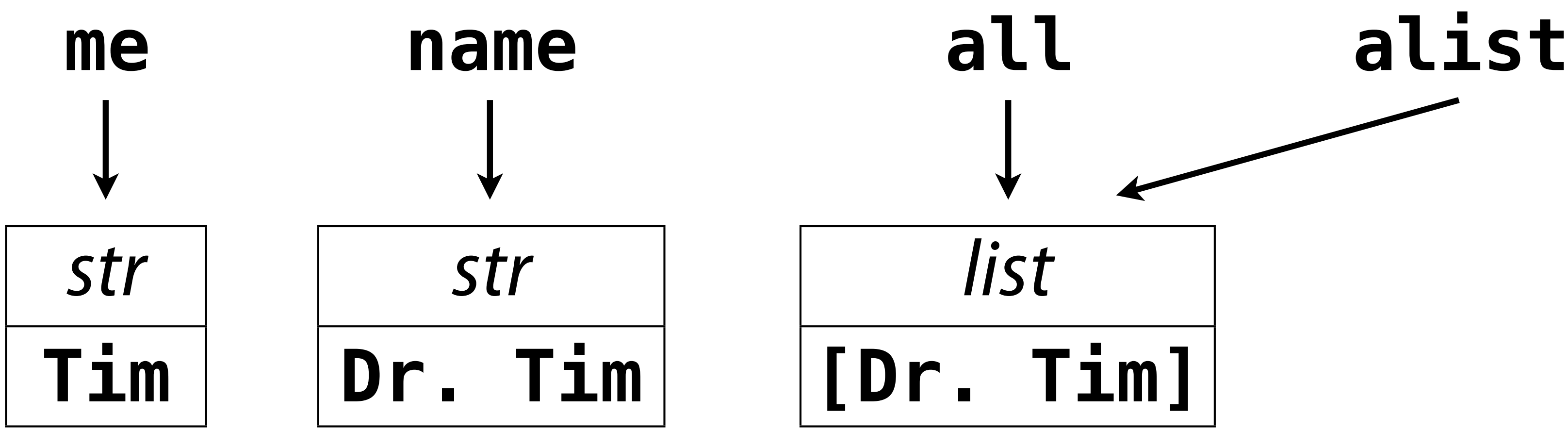
Arguments Are *Assigned*

```
def addp(name, alist):
    name = 'Dr. ' + name
    alist.append(name)

me = 'Tim'
all = []
addp('Tim', all)
print me, all
```



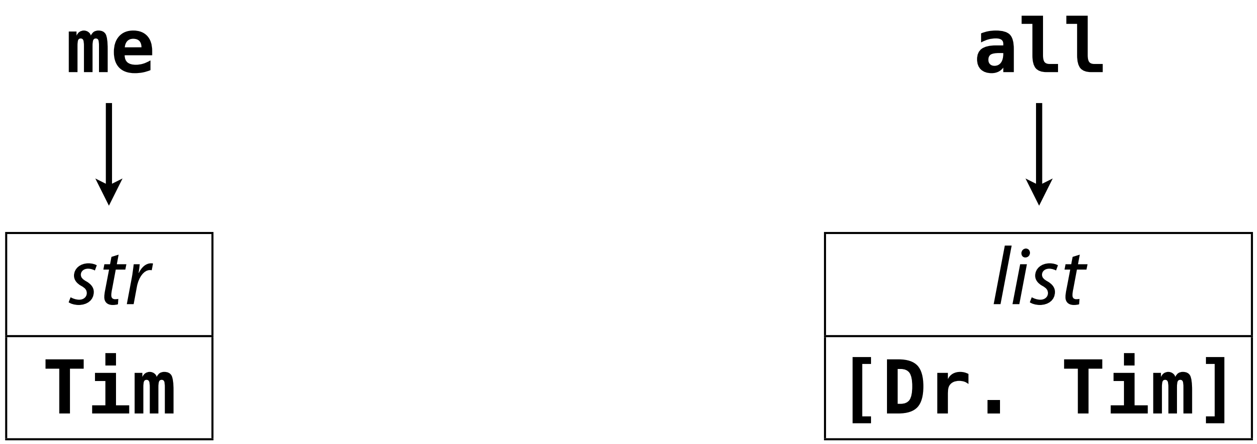
Arguments Are *Assigned*

```
def addp(name, alist):
    name = 'Dr. ' + name
    alist.append(name)

me = 'Tim'
all = []
addp('Tim', all)
print me, all
```



Arguments Are ***Assigned***

```
def addp(name, alist):
    name = 'Dr. ' + name
    alist.append(name)

me = 'Tim'
all = []
addp('Tim', all)
print me, all
```



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
foo()                         => TypeError
```

- 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

Namespace Interlude

Namespace

```
a = 42
b = 'foo'
c = ['a']
```

Name	Object
`'a'`	`int:42`
`'b'`	`str:'foo'`
`'c'`	`list:['a']`

- Maps from (variable) name (string) to object
- What does this remind you of?
- Look at `__dict__` attributes sometime…

Nested Namespaces

- Like data structures, namespaces can be nested
- How to create nested namespaces? We will see…
- Typically, access nested namespaces with dot (.)

Name	Object
'a'	int:42
'foo'	*namespace*
'c'	list:['a']

Name	Object
'a'	str:'wibble'
'b'	int:42
'c'	float:3.14

```
print a         # 42
print foo.a     # 'wibble'
```

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)`

Using a Class

- Classes and objects are namespaces!

```
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
- Write three functions (that I specify)

- BE SURE TO LABEL YOUR PRINTOUT!!!

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

"""Homework for CS 368-2 (2012 Spring)
Assigned on Day 05, 2012-03-27
Written by <Your Name>
"""
```