Outline

Variable Names and References

Assignment Statements

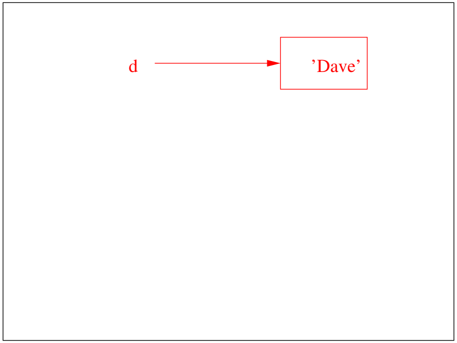
An assignment statement in Python associates an object with the name of a variable.

More precisely, the name is associated with a reference to an object of some class, or type.

This association remains in effect until a new object reference is associated with the variable through a new assignment statement.

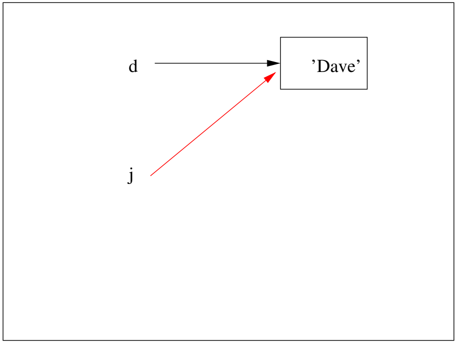
Python Assignment Examples

```
d = 'Dave'
j = d
j = 'John'
d = 'Smith'
```



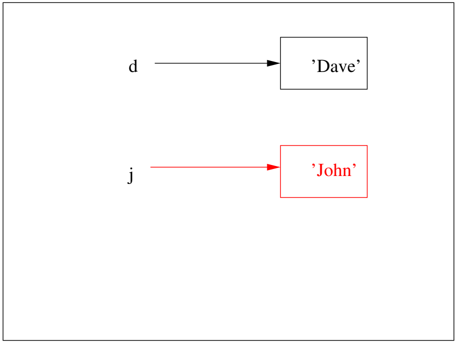
Python Assignment Examples

```
d = 'Dave'
j = d
j = 'John'
d = 'Smith'
```



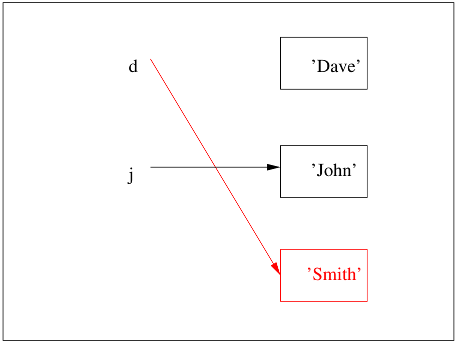
Python Assignment Examples

```
d = 'Dave'
j = d
j = 'John'
d = 'Smith'
```



Python Assignment Examples

```
d = 'Dave'
j = d
j = 'John'
d = 'Smith'
```



Namespaces

The Local Dictionary

The values of local variables–those which are currently active, are kept by Python, along with function names, in a dictionary object, called a namespace.

This dictionary is available by calling the built-in function `locals()`.

The Python function `id` returns a unique identifier for each data object (i.e. memory address where the object is stored)

Namespace can be modified directly. For example, the command `del d` removes the name 'd' from the local namespace.

See example program `Section4_2.py`.

Variable Types and References in Python

Dynamic Typing

In Python's memory model a variable just contains a reference to an object.

All variables have the same size, i.e. the standard address size of the computer, usually 4 or 8 bytes.

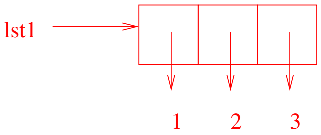
The data type information is stored with the object.

In order to avoid confusion with other languages, some people prefer to talk about **names**, rather than use traditional term **variable**.

The term dynamic typing in Python means that a variable/name can refer to objects of different types.

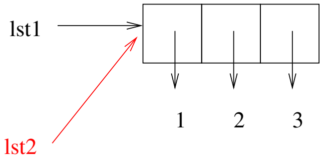
Aliasing

```
lst1 = [1, 2, 3]
lst2 = lst1
lst2.append(4)
lst1
```



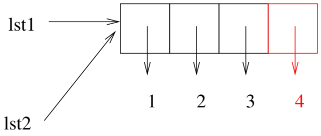
Aliasing

```
lst1 = [1, 2, 3]
lst2 = lst1
lst2.append(4)
lst1
```



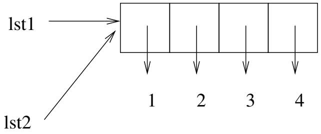
Aliasing

```
lst1 = [1, 2, 3]
lst2 = lst1
lst2.append(4)
lst1
```



Aliasing

```
lst1 = [1, 2, 3]
lst2 = lst1
lst2.append(4)
lst1
[1, 2, 3, 4]
```



Copying Variable Values in Python

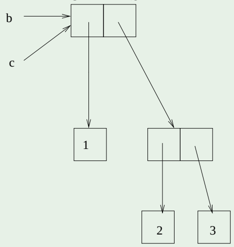
Deep and Shallow Copy

To avoid the problem of aliasing, we can, by using the copy function, force a new copy of a value to be created, so when it gets changed, the original variable, which refers to the original object, will keep its original value.

Copying Variable Values in Python

Deep and Shallow Copy

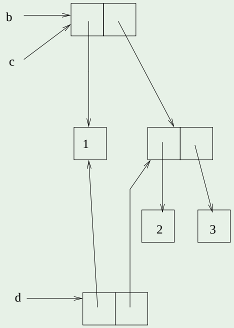
```
 >>>from copy import *
>>> b = [1, [2, 3]]
>>> c = b
```



Copying Variable Values in Python

Shallow Copy

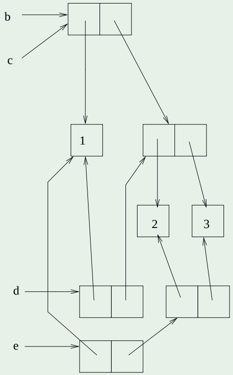
```
>>> d = copy(b)
```



Copying Variable Values in Python

Deep Copy

If a container object refers to other objects, these can be copied as well, using the deepcopy function. `>>> e = deepcopy(b)`



Passing Parameters in Function Calls

Formal vs. Actual Parameters

Formal parameters are the variable names used in implementing the function. They are listed in parentheses after the function name in the function definition, then used to express the Python commands needed to perform the algorithm of the function.

Actual parameters are the variable names used by the program where the function is called with specific values. When it is called, the function cannot change the value of an actual parameter, but it can change an object to which the actual parameter refers.

Passing Parameters in Function Calls

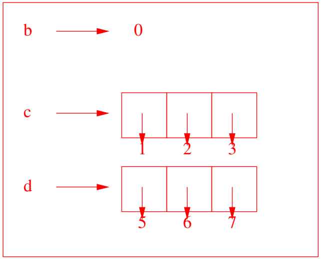
Example

Example:

```
  def f1(a,b):        a and b are formal parameters
   """ pre:  a and b are integers
       post:  returns sum and product of a and b"""

   return a+b, a*b

c,d = 1,-9
sum,prod = f1(c,d)       c and d are actual parameters
```

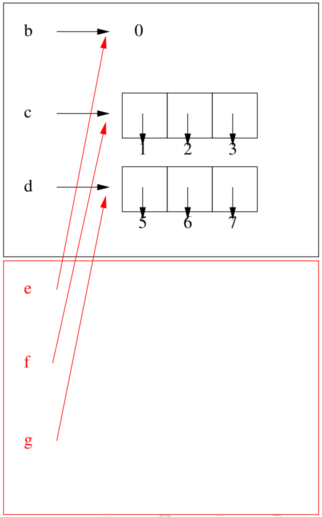
A Function Call Example

```
def func(e, f, g):
    e += 2
    f.append(4)
    g = [8, 9]
    print(e, f, g)

def main():
    b = 0
    c = [1, 2, 3]
    d = [5, 6, 7]
    func(b, c, d)
    print(b, c, d)
```



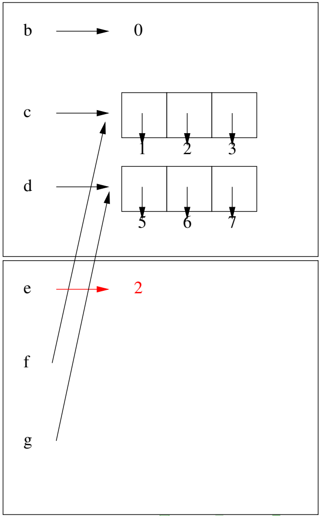
A Function Call Example

```
def func(e, f, g):
    e += 2
    f.append(4)
    g = [8, 9]
    print(e, f, g)

def main():
    b = 0
    c = [1, 2, 3]
    d = [5, 6, 7]
    func(b, c, d)
    print(b, c, d)
```



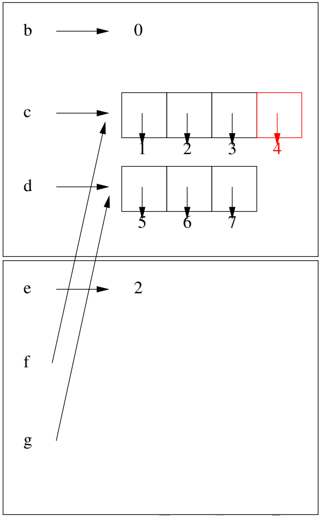
A Function Call Example

```
def func(e, f, g):
    e += 2
    f.append(4)
    g = [8, 9]
    print(e, f, g)

def main():
    b = 0
    c = [1, 2, 3]
    d = [5, 6, 7]
    func(b, c, d)
    print(b, c, d)
```



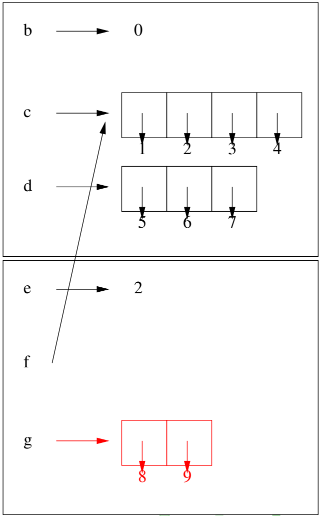
A Function Call Example

```
def func(e, f, g):
    e += 2
    f.append(4)
    g = [8, 9]
    print(e, f, g)

def main():
    b = 0
    c = [1, 2, 3]
    d = [5, 6, 7]
    func(b, c, d)
    print(b, c, d)
```



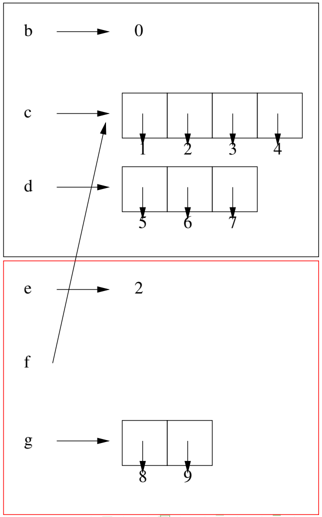
A Function Call Example

```
def func(e, f, g):
    e += 2
    f.append(4)
    g = [8, 9]
    print(e, f, g)

def main():
    b = 0
    c = [1, 2, 3]
    d = [5, 6, 7]
    func(b, c, d)
    print(b, c, d)
```



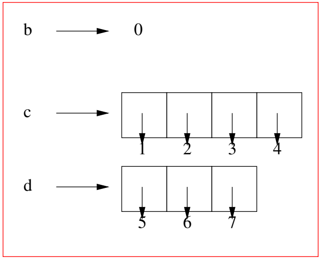
A Function Call Example

```
def func(e, f, g):
    e += 2
    f.append(4)
    g = [8, 9]
    print(e, f, g)

def main():
    b = 0
    c = [1, 2, 3]
    d = [5, 6, 7]
    func(b, c, d)
    print(b, c, d)
2 [1, 2, 3, 4] [8, 9]
```



A Function Call Example

```
def func(e, f, g):
   e += 2
   f.append(4)
   g = [8, 9]
   print(e, f, g)

 def main():
   b = 0
   c = [1, 2, 3]
   d = [5, 6, 7]
   func(b, c, d)
   print(b, c, d)
 2 [1, 2, 3, 4] [8, 9]
 0 [1, 2, 3, 4] [5, 6, 7]
```



A Function Call Example

Keep in mind

- a function can change the state of an object that actual parameter refers to.
- a function cannot change which object the actual parameter refers to.
- assigning a new object to a formal parameter inside a function or method will never change the actual parameter in any way, regardless of whether the actual parameter is mutable or not.

A Weakness of Using Arrays to Implement A List ADT

Problem

from previous class: Python uses arrays of references to implement the list ADT
(*arrays can easily be traversed by proceeding along a series of contiguous memory locations; arrays allow random access: jumping quickly to any index location in the array, according to a formula for the address which is easy to calculate.*)
insert and delete operations for lists were both $\Theta(n)$
(*they both require copying much of the list to keep its sequential order*)
For long lists, this is a problem.

Another design for sequential lists: it is possible to get faster insert and delete operations at the cost of slower random access.

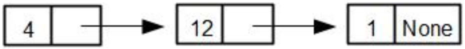
Linked Lists

Data Items As Nodes

Using the idea of a reference (pointer), ordering items into a list can be accomplished by having each item have its own reference to the next item.

The list can be traversed sequentially by hopping from each item to the one it refers to.

The value of each item is kept together with the reference/pointer to the next item in an object called a node.



Linked Lists

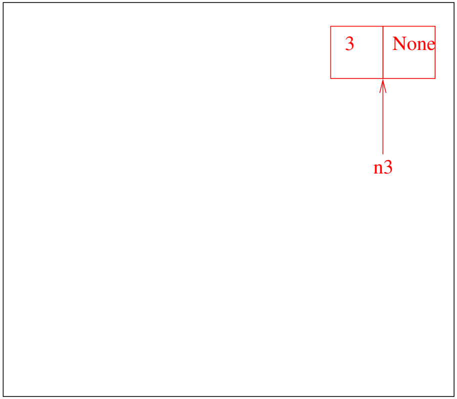
The ListNode Class

We can define a class to support this structure, with attributes `item` (for the data value) and `link` (for the reference to the next item):

```
class ListNode:
  def __init__(self, item = None, link = None):
    """
    post:  creates a ListNode with the
    specified data value and link
    """
    self.item = item
    self.link = link
```

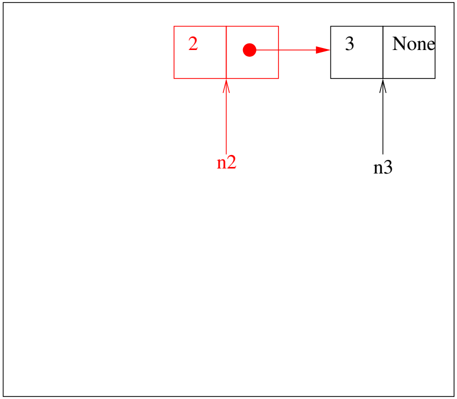
Building a Linked List

```
n3 = ListNode(3)
```



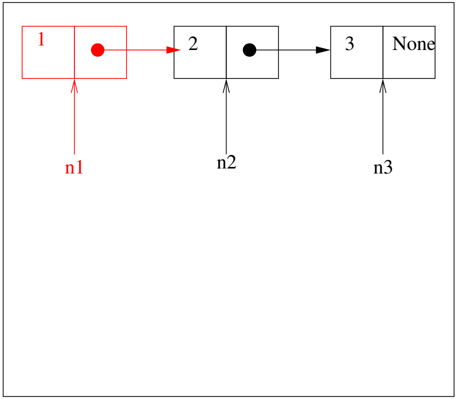
Building a Linked List

```
n3 = ListNode(3)
n2 = ListNode(2, n3)
```



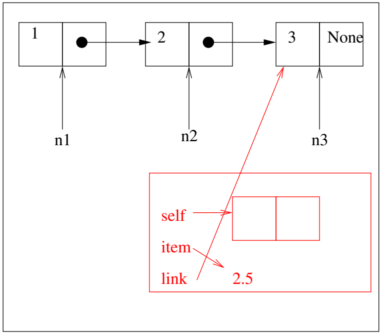
Building a Linked List

```
n3 = ListNode(3)
n2 = ListNode(2, n3)
n1 = ListNode(1, n2)
```



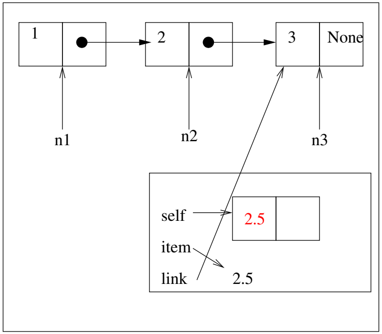
Inserting a Node Into A Linked List

```
def __init__(self, item,
link):
   self.item = item
   self.link = link
n25 = ListNode(2.5, n2.link)
n2.link = n25
```



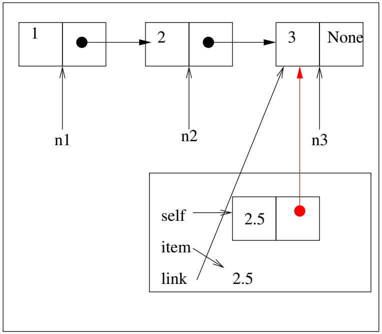
Inserting a Node Into A Linked List

```
def __init__(self, item,
link):
   self.item = item
   self.link = link
n25 = ListNode(2.5, n2.link)
n2.link = n25
```



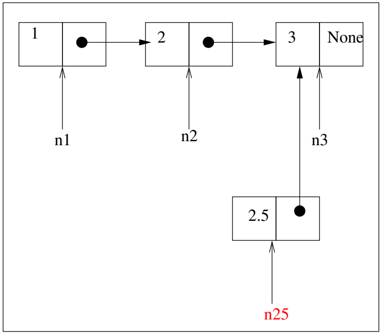
Inserting a Node Into A Linked List

```
def __init__(self, item,
link):
   self.item = item
   self.link = link
n25 = ListNode(2.5, n2.link)
n2.link = n25
```



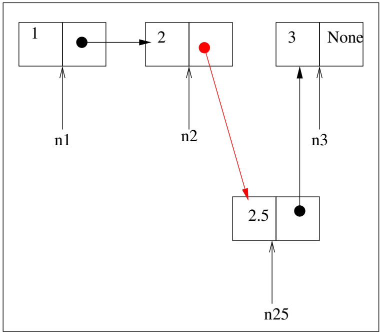
Inserting a Node Into A Linked List

```
def __init__(self, item,
link):
   self.item = item
   self.link = link
n25 =ListNode(2.5, n2.link)
n2.link = n25
```



Inserting a Node Into A Linked List

```
def __init__(self, item,
link):
   self.item = item
   self.link = link
n25 = ListNode(2.5, n2.link)
n2.link = n25
```



In-class work - part I

1. Add a method (or override __str__) to display the value of the node to the ListNode class (no need to display the reference to the next node)

2. Consider the following block of code:

```
n1 = ListNode(25)
n2 = ListNode(7)
n1.link = n2
n3 = ListNode(3)
n2.link = n3
```

Do the following:

– add the code to add 5 to the end of 'linked nodes collection',

– add the code to insert value 30 between 3 and 5, then

– add the code to delete node with 7.

Question: is there a more 'elegant way' of creating the 'linked nodes collection' of 25, 7, 3, and 5?

In-class work - part II

3. Find the asymptotic running time of the following procedure:

```
n = input("Enter an integer number greater than 5:")
for i in range(n):
   for j in range(n//10):
      print("i = ", i, ", j = ", j)
```

4. Look at the code of Sections4_1-4_3InClassWork.py and draw the memory model after each line of code.
5. Look at the code of Section4_2example2.py and draw the memory model after each line of code.