CSI33 Data Structures

Department of Mathematics and Computer Science
Bronx Community College

September 18, 2017

Outline

Outline

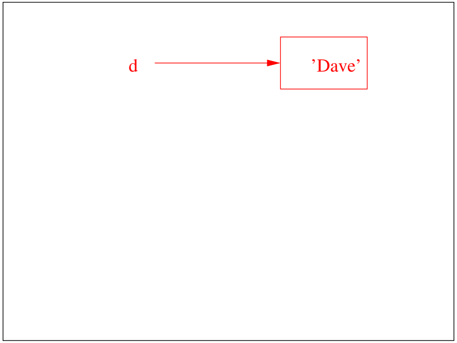
Variable Names and References

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.

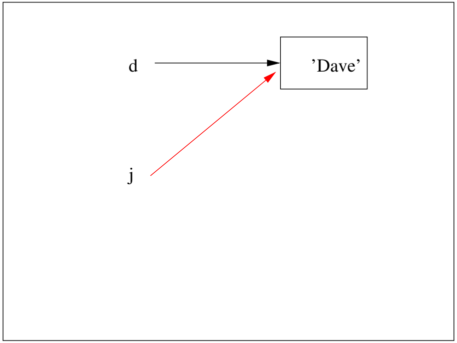
Python Assignment Examples

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



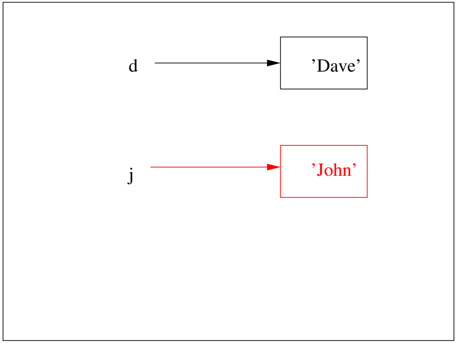
Python Assignment Examples

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



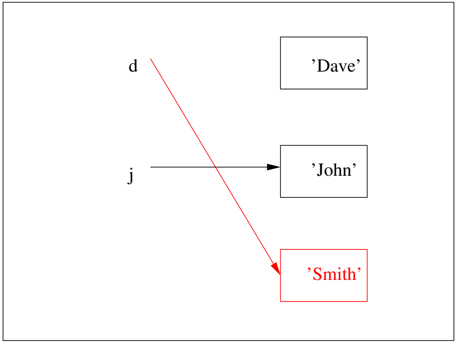
Python Assignment Examples

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

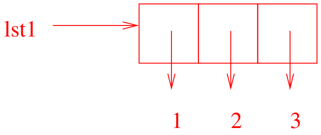
Variable Types and References in Python

Dynamic Typing

Dynamic typing in Python means that a variable does not have a fixed type like `int`. Rather, its type can change, depending on what object it refers to for its value. This is because the referent (the object referred to) contains the type information for that value. If the variable is assigned to another object, it will then have the type of the other object.

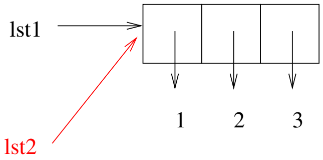
Aliasing

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



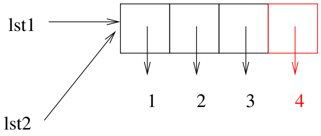
Aliasing

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



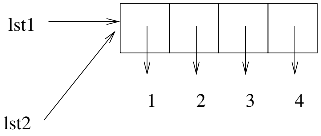
Aliasing

```
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

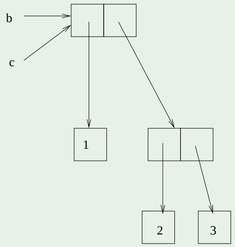
Deep and Shallow Copy

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

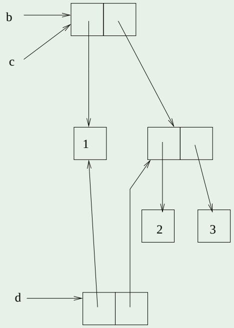
```
 >>>from copy import *
>>> b = [1, [2, 3]]
>>> c = b
```



Copying Variable Values in Python

Shallow Copy

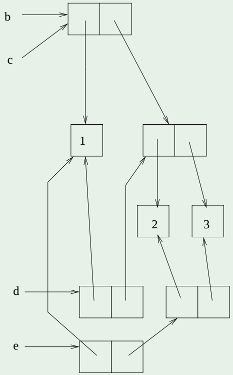
```
>>> d = copy(b)
```



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

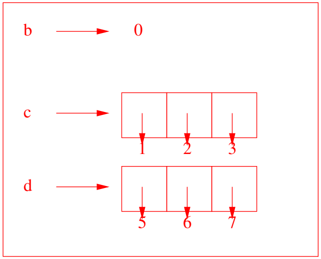
Formal vs. Actual Parameters

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.

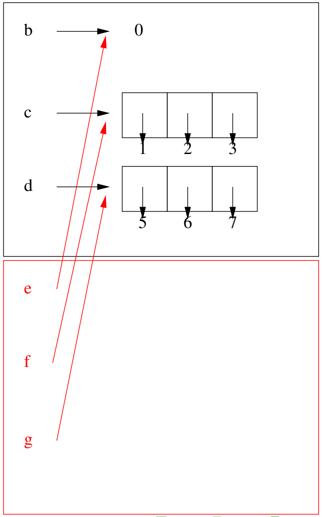
A Function Call Example

```
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)
```



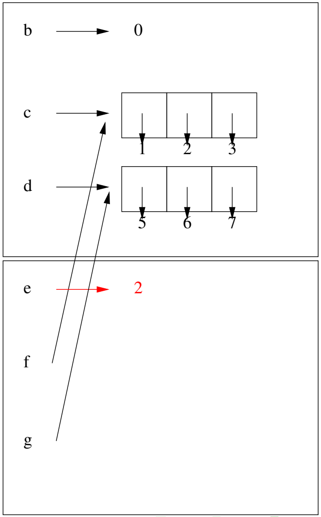
A Function Call Example

```
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)
```



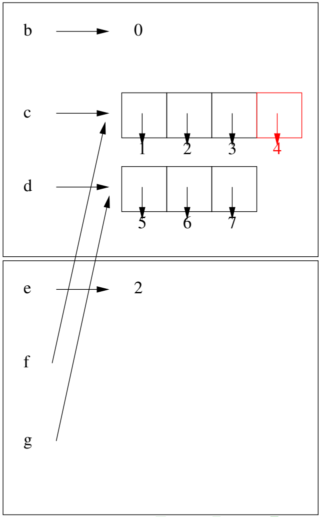
A Function Call Example

```
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)
```



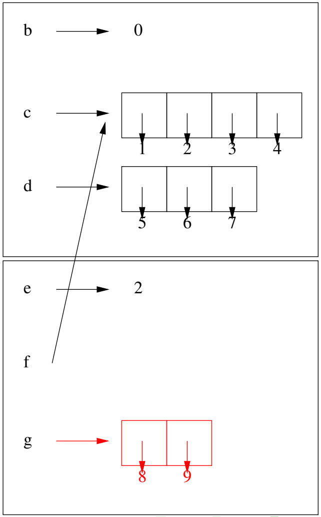
A Function Call Example

```
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)
```



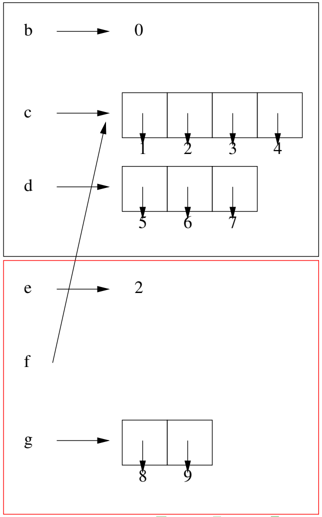
A Function Call Example

```
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)
```



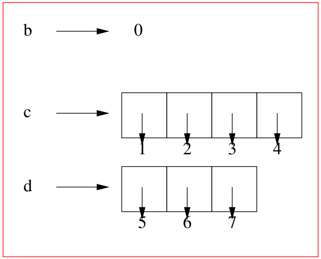
A Function Call Example

```
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 Weakness of Using Arrays to Implement A List ADT

Problem

Python uses arrays of references to implement the list ADT, because arrays can easily be traversed by proceeding along a series of contiguous memory locations. Arrays also allow random access, that is, jumping quickly to any index location in the array, according to a formula for the address which is easy to calculate. But using arrays, the insert and delete operations for lists were both $\Theta(n)$, since they both require copying much of the list to keep its sequential order. For long lists, this is a problem. Fortunately, another design for sequential lists is possible which has 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

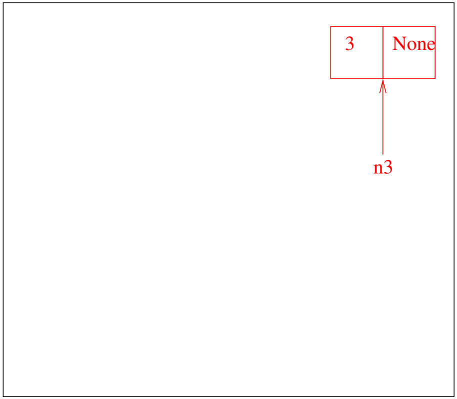
The ListNode Class

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

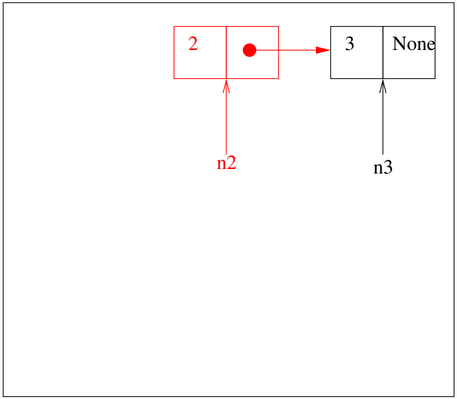
Building A Linked List

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



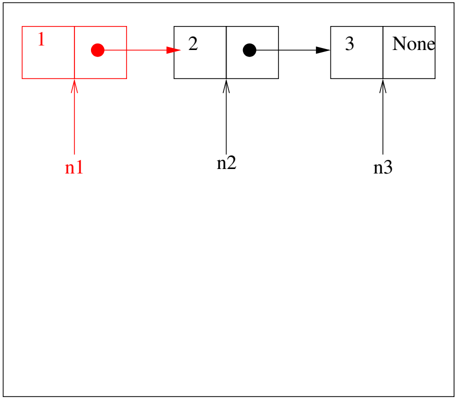
Building a Linked List

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



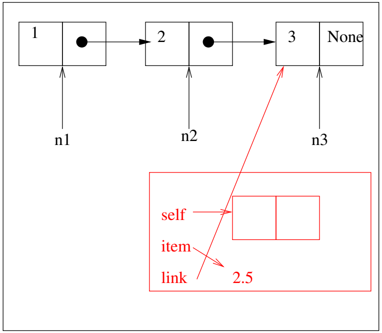
Building a Linked List

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



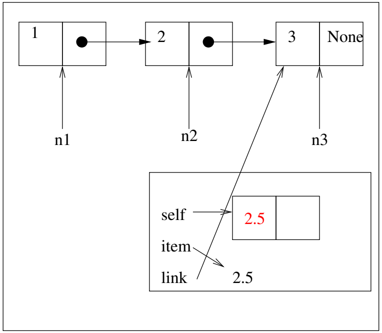
Inserting a Node Into A Linked List

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



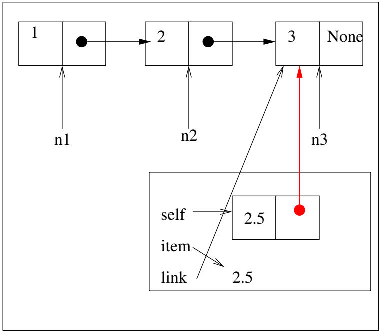
Inserting a Node Into A Linked List

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



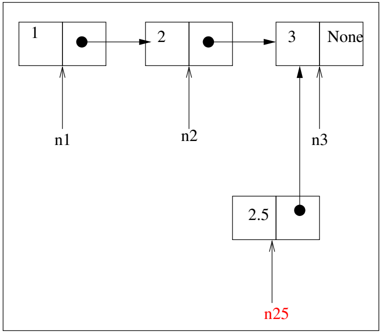
Inserting a Node Into A Linked List

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



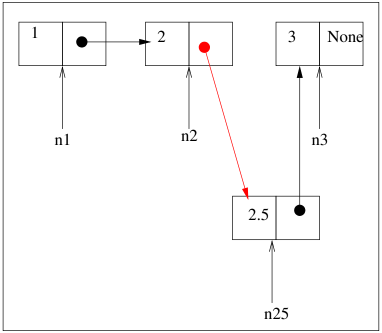
Inserting a Node Into A Linked List

```
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
```


1
2
3
None
n1
n2
n3
2.5
n25