Understanding Python Referencing

Binding a variable in Python means setting a **name** to hold a **reference** to some **object**.

A reference is **deleted** via **garbage collection** after any names bound to it have passed out of scope.

You can also assign to multiple names at the same time.

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

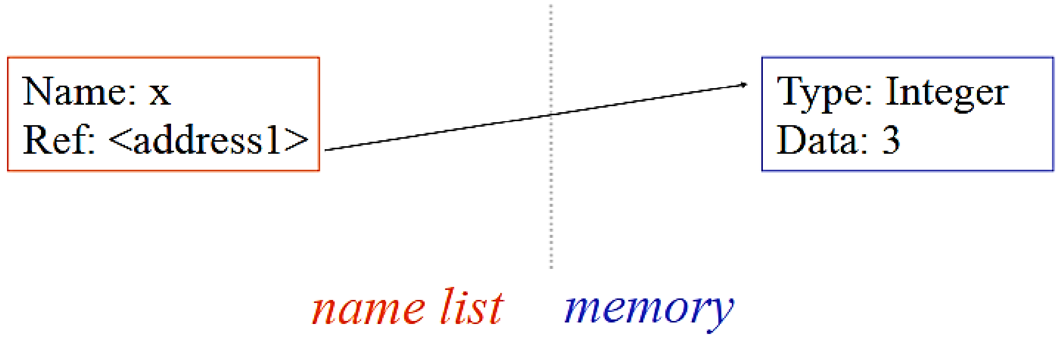
Understanding Python Referencing

Assignment manipulates references.

- x = y **does not make a copy** of the object y references. It makes x reference the object y references

What happens when you type x = 3 ?

- First, an integer 3 is created and stored in memory
- A name x is created
- A reference to the memory location storing the 3 is then assigned to the name x
- So, when we say that the value of x is 3, we mean that x now refers to the integer 3



Understanding Python Referencing

A mutable object can be changed after it is created, and an immutable object can't.

Objects of built-in types like (**int**, **float**, **bool**, **str**, **tuple**, **unicode**) are **immutable**. Objects of built-in types like (**list**, **set**, **dict**) are **mutable**.

This doesn't mean we can't change the value of x, i.e. change what x refers to.

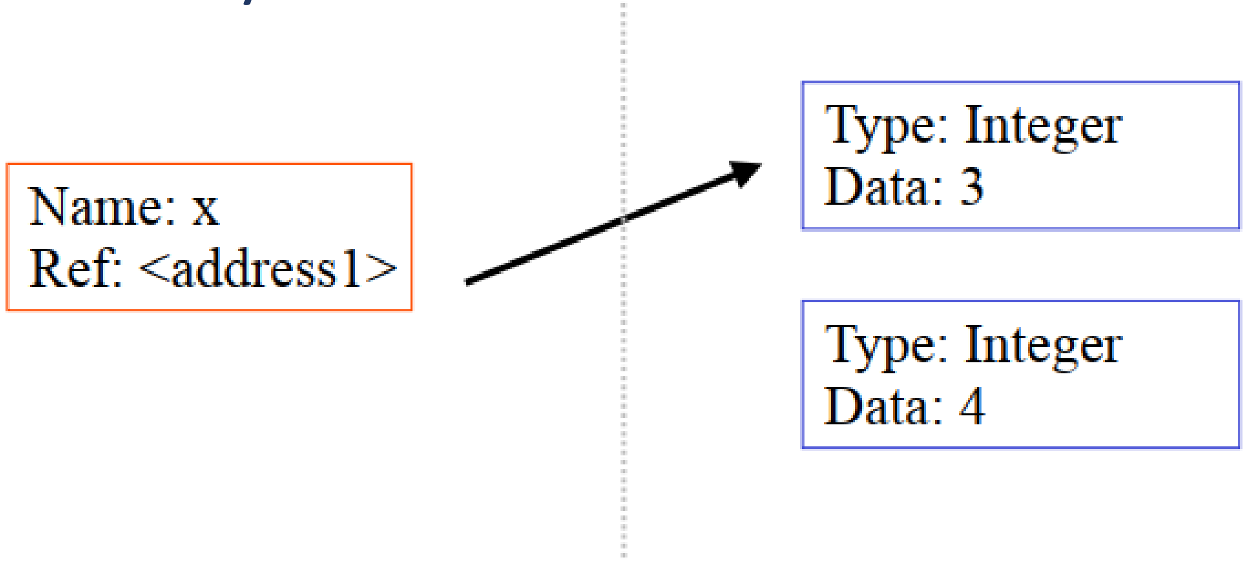
For example, we could increment x:

```
>>> x = 3
>>> x = x + 1
>>> print x
4
```

Understanding Python Referencing

If we increment x, then what's really happening is:

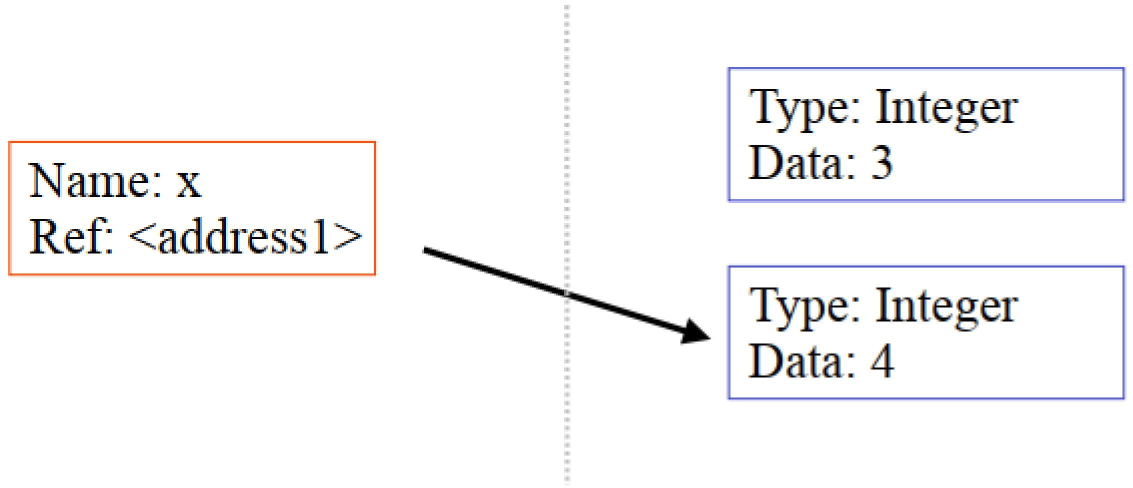
- The reference of name x is looked up.
- The value at that reference is retrieved.
- The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.



Understanding Python Referencing

If we increment x, then what’s really happening is:

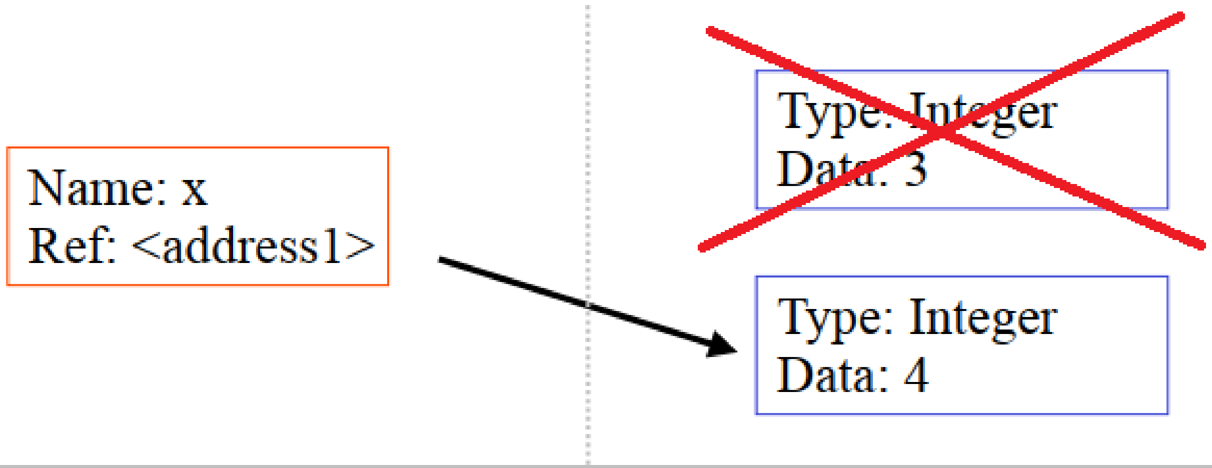
- The reference of name x is looked up.
- The value at that reference is retrieved.
- The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
- The name x is changed to point to this new reference.



Understanding Python Referencing

If we increment x, then what’s really happening is:

- The reference of name x is looked up.
- The value at that reference is retrieved.
- The 3+1 calculation occurs, producing a new data element 4 which is assigned to a fresh memory location with a new reference.
- The name x is changed to point to this new reference.
- The old data 3 is garbage collected if no name still refers to it



Understanding Python Referencing

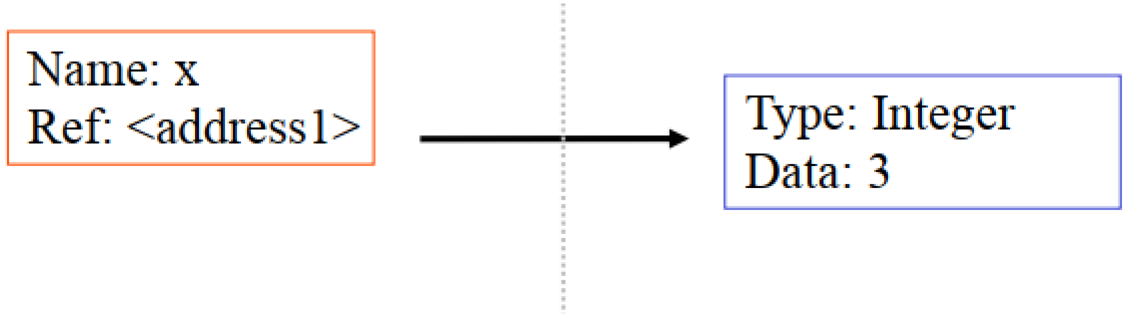
So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

```
>>> x = 3     # Creates 3. Name x refers to 3
>>> y = x     # Creates name y, refers to 3
>>> y = 4     # Creates reference for 4. Changes y
>>> print x   # No effect on x. Still refers to 3


3
```

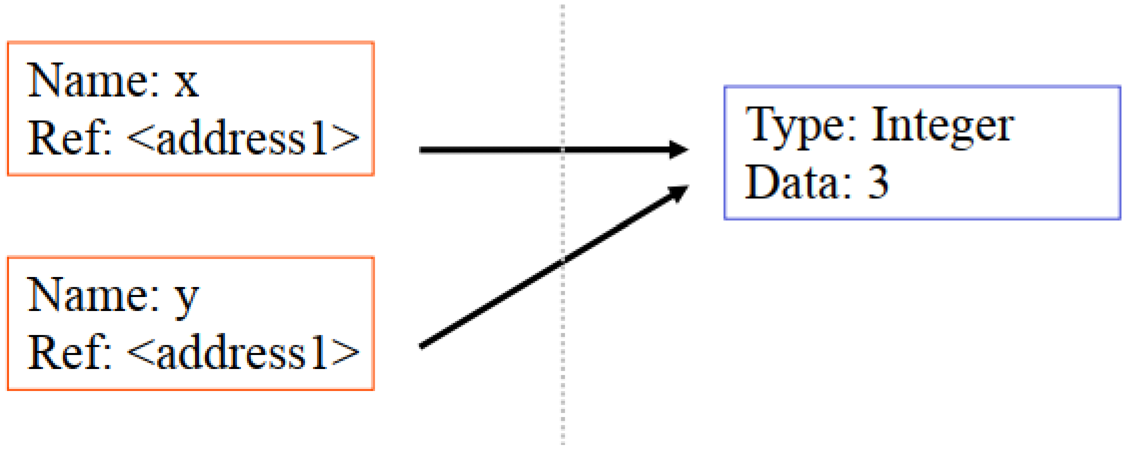
Understanding Python Referencing

So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

```
>>> x = 3   # Creates 3. Name x refers to 3
```



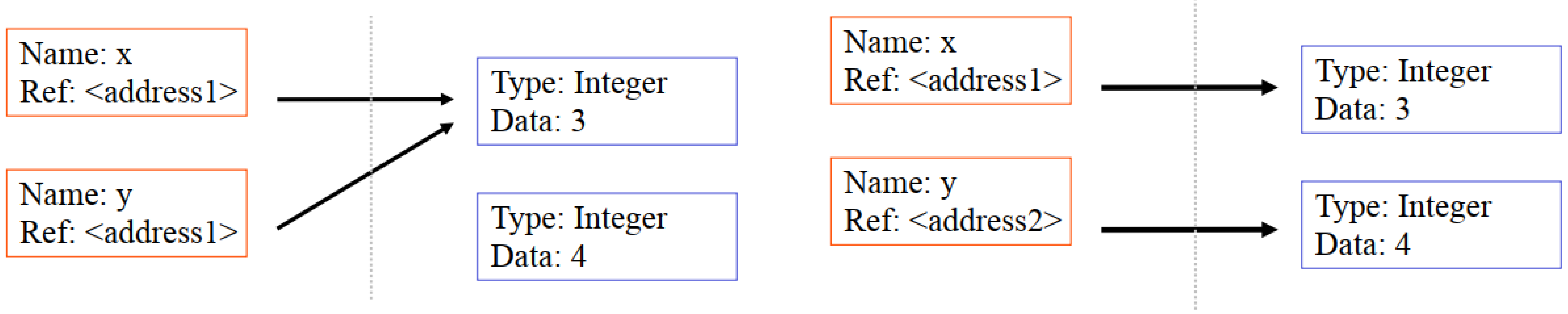
```
>>> y = x   # Creates name y, refers to 3
```



Understanding Python Referencing

So, for simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

```
>>> y = 4   # Creates reference for 4. Changes y
```



```
>>> print x # No effect on x. Still refers to 3
3
```

Understanding Python Referencing

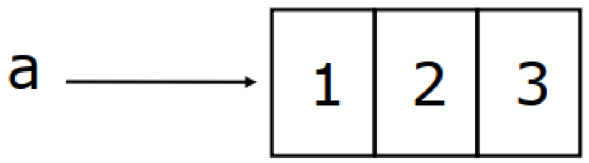
For other data types (lists, dictionaries, user-defined types), assignment works differently.

- These datatypes are “mutable.”
- When we change these data, we do it in place.
- We don’t copy them into a new memory address each time.
- If we type y = x and then modify y, both x and y are changed.

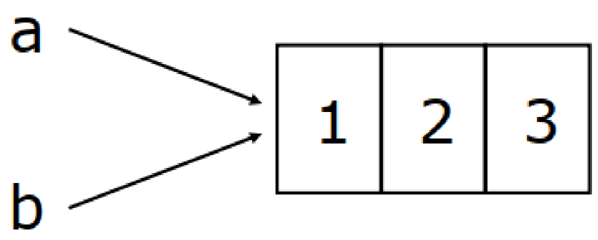
```
>>> a = [1,2,3]     # a now references the list [1,2,3]
>>> b = a          # b now references what a references
>>> a.append(4)     # this changes the list that a references
>>> print b        # we print what b references

[1,2,3,4]                  # WHY?
```

Understanding Python Referencing

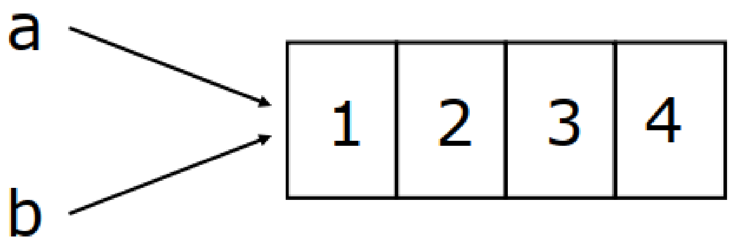
```
>>> a = [1,2,3]
```



```
>>> b = a
```



```
>>> a.append(4)
```



```
>>> print b
[1,2,3,4]
```

Python Collections (Arrays)

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.
- **Dictionary** is a collection which is ordered** and changeable. No duplicate members
- Strings: Immutable, Conceptually very much like a tuple

List

A list in Python is used to store the sequence of various types of data. Python lists are mutable type it mean we can modify its element after it created.

Unlike arrays in other programming languages, lists can store heterogeneous items

Can be written as a list of comma-separated values (items) between square brackets.

```
Height = [1.73 , 1.68, 1.76 ]  #Storing heights of persons in Height

#you want to store height and name both then
 Height = [“rahul”, 1.75, “person”, 1.63, “person2”, 1.96]
```

Nesting is possible

```
Person_heights = [[“person1”, 1.75] , [“person2”, 1.68]]
```

Characteristics of Lists

The list has the following characteristics:

- The lists are ordered.
- The element of the list can access by index.
- The lists are the mutable type.
- A list can store the number of various elements.

List

- List items can be accessed by indices which starts from zero **0** for the first element and then continues incrementally.

```
print(Height[0])
```

- Negative indices apply
- List are mutable as we seen they can be accessed by index we can also manipulate them by assignment operator

Example

```
Values = [ 1, 2, 3, 4,5, 6]
Values[0] = 9
print(Values)
```

Output

```
 [9, 2, 3, 4, 5, 6]
```

Allow Duplicates

```
thislist = ["apple", "banana", "cherry", "apple", "cherry"]
print(thislist)
```

List Length

To determine how many items a list has, use the len() function:

```
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
```

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
list4 = ["abc", 34, True, 40, "male"]
print(list4)
print(type(list1))
print(list1[1])
```

- Negative Indexing

Negative indexing means start from the end

-1 refers to the last item, -2 refers to the second last item etc.

```
thislist = ["apple", "banana", "cherry"]
print(thislist[-1])
```

List

- We can use slicing operator **[n:m]** to get a subset list or a particular item.

- The **:** symbol is used for more than one purpose in Python.

- As **slice** operator with sequences

- The : operator slices a part from a sequence object such as list, tuple or string.
- It takes two arguments. First is the index of start of slice and second is index of end of slice.
- Both operands are optional. If first operand is omitted, it is 0 by default. If second is omitted, it is set to end of sequence.

List

Example :

```
b= [5,10,15,20,25]
#printing upto 3 indices
print(b[:3])     #b[0:3}
#reversing a list with use of slicing
print(b[::-1])
#printing from indices 1 to 3
print(b[1:3])
```

Output

```
 [5, 10, 15]
[25, 20, 15, 10, 5]
[10, 15]
```

- By leaving out the start value, the range will start at the first item:

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[:4])
```

Output: ['apple', 'banana', 'cherry', 'orange']

- By leaving out the end value, the range will go on to the end of the list:

```
thislist = ["apple","banana","cherry","orange","kiwi","melon","mango"]
print(thislist[2:])
```

Output: ['cherry', 'orange', 'kiwi', 'melon', 'mango']

Check if Item Exists

To determine if a specified item is present in a list use the **in** keyword:

```
l=["apple","banana","cherry"]
if(“apple" in l):
 print("Yes,'apple' is in the fruits list")
```

Change Item Value

To change the value of a specific item, refer to the index number:

```
thislist=["apple","banana","cherry"]
thislist[1]="blackcurrant"
print(thislist)
```

If you insert *more* items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly:

```
thislist = ["apple", "banana", "cherry"]
thislist[1:2] = ["blackcurrant", "watermelon"]
print(thislist)
```

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

Delete List Elements

```
list1 = ['physics', 'chemistry', 1997, 2000];
print(list1)
del list1[2];
print("After deleting value at index 2 : ")
print(list1)
```

Python list method **remove()** searches for the given element in the list and removes the first matching element.

```
list.remove(obj)
```

Parameters: obj − This is the object to be removed from the list.

```
aList = [123, 'xyz', 'zara', 'abc', 'xyz']
aList.remove('xyz')
print ("List : ", aList)
aList.remove('abc')
print ("List : ", aList)
```

Homework:

- Write atleast 10 Built-in List Functions & Methods with example.
- **Difference between Append, Extend and Insert**

```python
# comparison between the three methods


# assign lists
list_1 = [1, 2, 3]
list_2 = [1, 2, 3]
list_3 = [1, 2, 3]


a = [2, 3]


# use methods
list_1.append(a)
list_2.insert(3, a)
list_3.extend(a)


# display lists
print(list_1)
print(list_2)
print(list_3)
print(len(list_1))
print(len(list_2))
print(len(list_3))
```

Output:

```
[1, 2, 3, [2, 3]]
[1, 2, 3, [2, 3]]
[1, 2, 3, 2, 3]


4
4
5
```