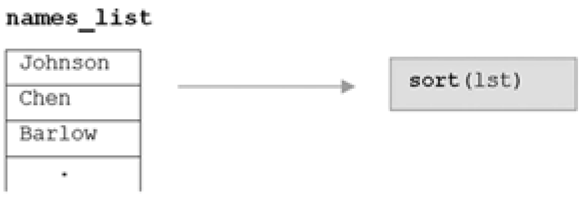
Software Objects

Objects are the fundamental component of object-oriented programming. All values in Python are represented as objects. This includes, for example, lists, as well as numeric values.

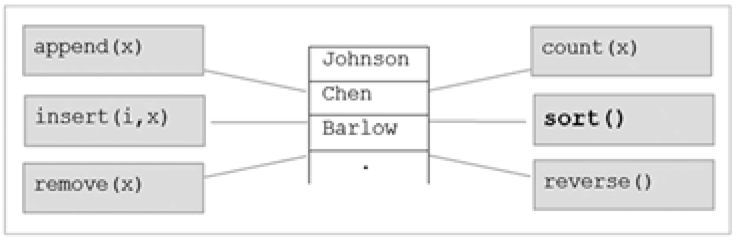
What Is an Object?

The notion of software objects derives from objects in the real world. All objects have certain *attributes* and *behavior*. The attributes of a car, for example, include its color, number of miles driven, current location, and so on. Its behaviors include driving the car (changing the number of miles driven attribute) and painting the car (changing its color attribute), for example.

Similarly, an **object** contains a set of attributes, stored in a set of **instance variables**, and a set of functions called **methods** that provide its behavior. For example, when sorting a list in procedural programming, there are two distinct entities—a sort function and a list to pass it, as seen below:



In object-oriented programming, the `sort` routine would be *part of* the object containing the list.

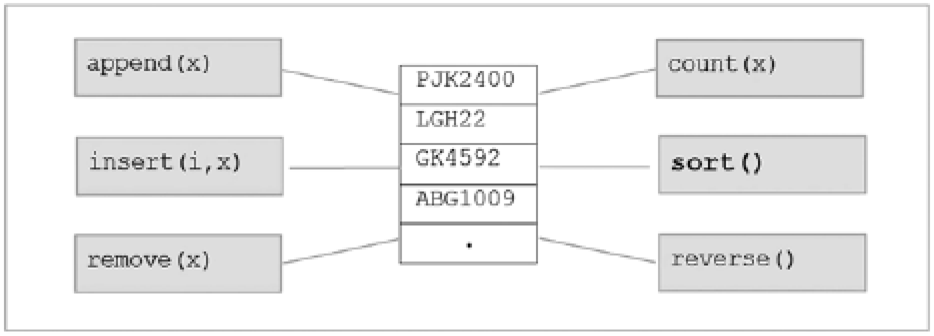


Here, `names_list` is an object instance of the Python built-in list type. All list objects contain the same set of methods. Thus, `names_list` is sorted by simply calling that object's `sort` method,

```
names_list.sort()
```

The period is referred to as the *dot operator*, used to select a member of a given object—in this case, the `sort` method. Note that no arguments are passed to `sort`. That is because methods

operate on the data of the object that they are part of. Thus, the `sort` method does not need to be told which list to sort.
Suppose there were another list object called `part_numbers`, containing a list of automobile part numbers. Since all list objects behave the same, `part_numbers` would contain the identical set of methods as `names_list`. The data that they would operate on, however, would be different. Thus, two objects of the same type differ only in the particular set of values that each holds.



In order to sort *this* list, therefore, the `sort` method of object `part_numbers` is called,

```
part_numbers.sort()
```

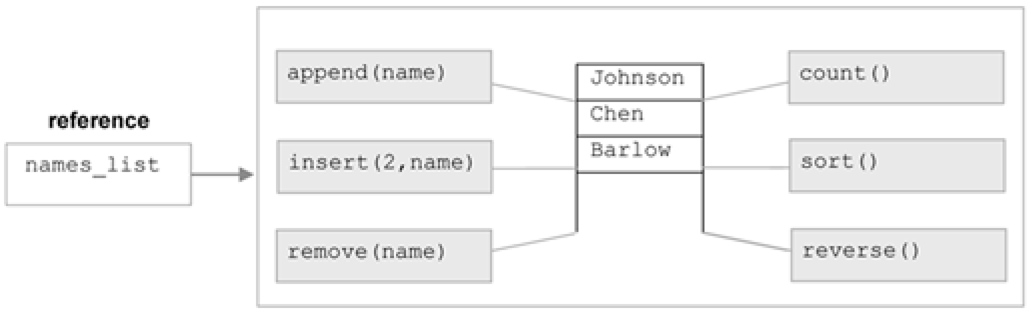
The `sort` routine is the same as the `sort` routine of object `names_list`. In this case, however, the list of part numbers is sorted instead. Methods `append`, `insert`, `remove`, `count`, and `reverse` also provide additional functionality for lists.

Part II - Object References

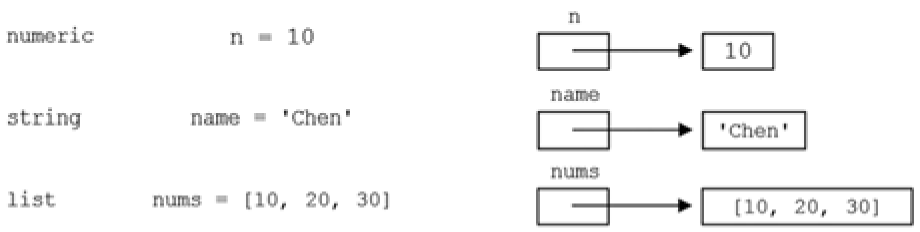
In this section we look at how objects are represented (which all values in Python are), and the effect it has on the operations of assignment and comparison, as well as parameter passing.

References in Python

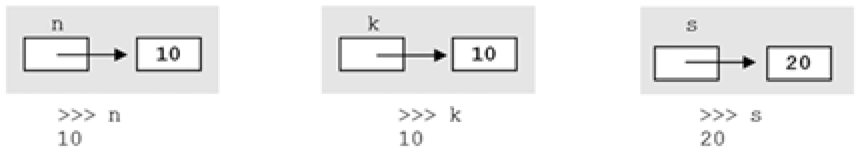
In Python, objects are represented as a *reference* to an object in memory, as seen below:



A **reference** is a value that references, or “points to,” the location of another entity. Thus, when a new object in Python is created, *two* entities are stored—the object, and a variable holding a reference to the object. All access to the object is through the reference value.



The value that a reference points to is called the **dereferenced value**. This is the value that the variable represents, as shown here:



We can get the reference value of a variable (that is, the location in which the corresponding object is stored) by use of **built-in function `id`**.

```
>>> id(n)        >>> id(k)        >>> id(s)
505498136        505498136        505498296
```

We see that the dereferenced values of `n` and `k`, `10`, is stored in the same memory location (`505498136`), whereas the dereferenced value of `s`, `20`, is stored in a different location (505498296). Even though `n` and `k` are each separately assigned literal value 10, they reference the *same instance* of `10` in memory (`505498136`). We would expect there to be separate instances of 10 stored. Python is using a little cleverness here. Since integer values are immutable, it assigned both `n` and `k` to the same instance. This saves memory and reduces the number of reference loca- tions that Python must maintain. From the programmer’s perspective however, they can be treated as if they are separate instances.

Your Turn

From the Python Shell, enter the following and observe the results.

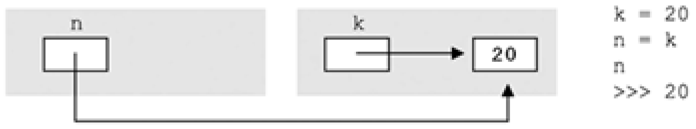
```
>>> n=10                             >>> n=20
>>> k=20                             >>> k=20

>>> id(n)                            >>> id(n)
???                                 ???

>>> id(k)                            >>> id(k)
???                                 ???
```

Part III - The Assignment of References

With our current understanding of references, consider what happens when variable n is assigned to variable k, as seen below:



```
k = 20
n = k
n
>>> 20
```

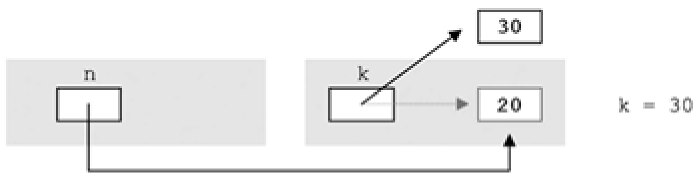
When variable n is assigned to k, it is the *reference value* of k that is assigned, not the dereferenced value 20, as shown below This can be determined by use of the built-in `id` function, as demonstrated below.

```
>>> id(k)              >>> id(k) == id(n)
505498136             True
```

```
>>> id(n)            >>> n is k
505498136             True
```

Thus, to verify that two variables refer to the same object instance, we can either compare the two `id` values by use of the comparison operator, or make use of the provided `is` operator (which performs `id(k) == id(n)`).
Thus, both `n` and `k` reference the same instance of literal value 20. This occurred in the above example when `n` and `k` were *separately* assigned 20 because integers are an immutable type, and Python makes attempts to save memory. In this case, however, `n` and `k` reference the same instance of 20 because assignment in Python assigns reference values. We must be aware of the fact, therefore, that when assigning variables referencing mutable values, such as lists, both variables reference the same list instance as well. We will discuss the implication of this next.

Finally, we look at what happens when the value of one of the two variables `n` or `k` is changed, as seen here:



Here, variable `k` is assigned a reference value to a *new* memory location holding the value `30`. The previous memory location that variable `k` referenced is retained since variable `n` is still referencing it. As a result, `n` and `k` point to different values, and therefore are no longer equal.

Your Turn

From the Python Shell, enter the following and observe the results.

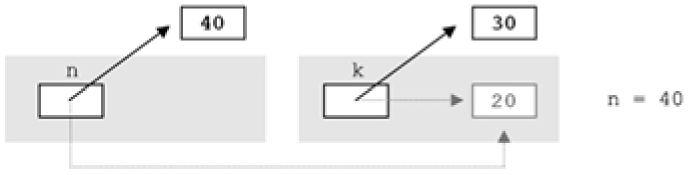
```
>>> n=10                            >>> n=20
>>> k=20                            >>> k=20


>>> id(n)                           >>> id(n)
???                                ???
```

```
>>> id(k)                              >>> id(k)
???                                   ???
```

Part III - Memory Deallocation and Garbage Collection

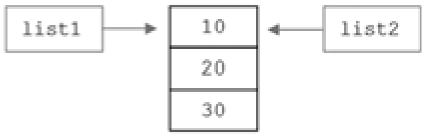
Next we consider what happens when in addition to variable `k` being reassigned, variable `n` is reassigned as well. The result is depicted below:



After `n` is assigned to `40`, the memory location storing integer value `20` is no longer referenced - thus, it can be *deallocated*. To **deallocate** a memory location means to change its status from “currently in use” to “available for reuse.” In Python, memory deallocation is automatically performed by a process called *garbage collection*. **Garbage collection** is a method of automatically determining which locations in memory are no longer in use and deallocating them. The garbage collection process is ongoing during the execution of a Python program.

List Assignment and Copying

Now that you understand the use of references in Python, we can revisit the discussion on copying lists from Chapter 4. We know that when a variable is assigned to another variable referencing a list, each variable ends up referring to the same instance of the list in memory, as seen below:



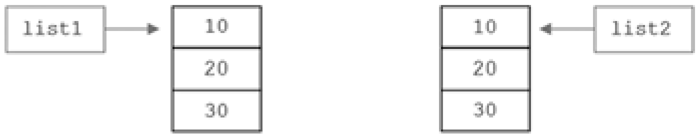
Thus, any changes to the elements of list1 results in changes to `list2`,

```
>>>list1[0] = 5
>>>list2[0]
5
```

We also learned that a copy of a list can be made as follows,

```
>>>list2 = list(list1)
```

`list()` is referred to as a *list constructor*. The result of the copying is illustrated below:.



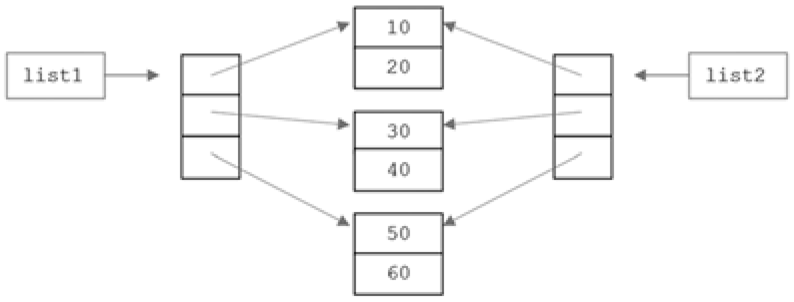
A copy of the list structure has been made. Therefore, changes to the list elements of `list1` will
not result in changes in `list2`.

```
>>>list1[0] = 5
>>>list2[0] 10
```

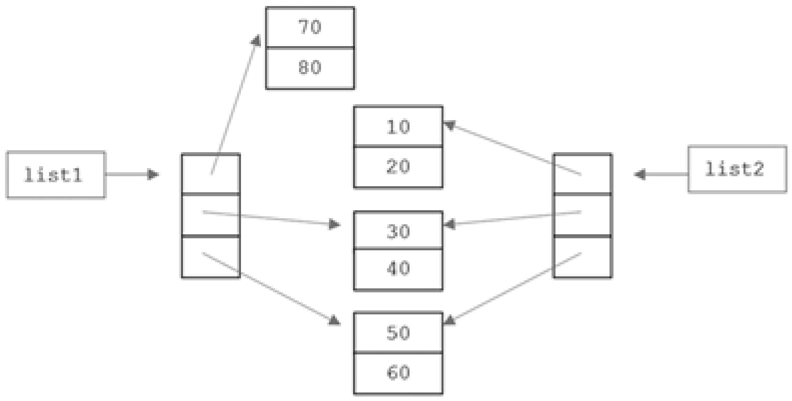
The situation is different if a list contains sublists, however.

```
>>>list1 = [[10, 20], [30, 40], [50, 60]]
>>>list2 = list(list1)
```

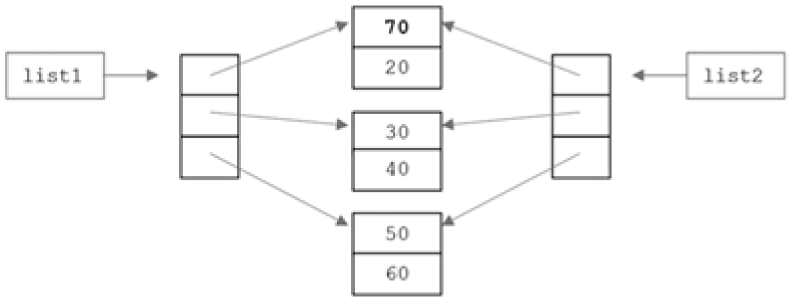
The resulting list structure after the assignment is shown here:



Notice that although copies were made of the top-level list structures, the elements *within* each list were not copied. This is referred to as a **shallow copy**. Thus, if a top-level element of one list is re-assigned, for example `list1[0] 5 [70, 80]`, the other list would remain unchanged, as seen below:



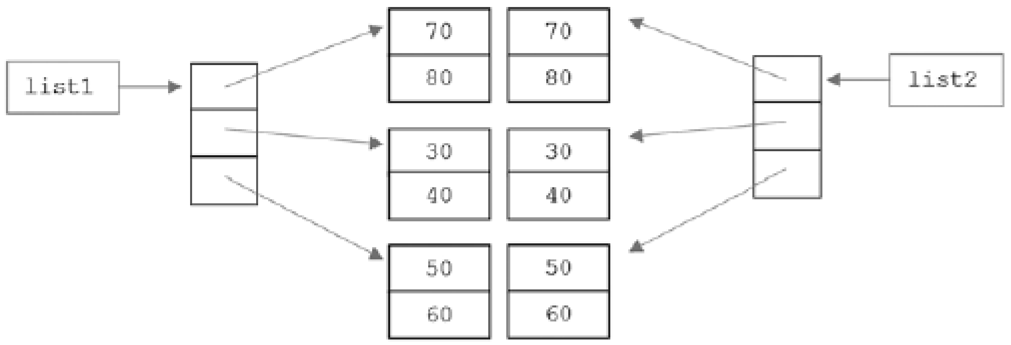
If, however, a change to one of the sublists is made, for example, `list1[0][0] 5 70`, the corresponding change would be made in the other list. That is, `list2[0][0]` would be equal to 70 also, as shown here:



A **deep copy** operation of a list (structure) makes a copy of the *complete* structure, including sub- lists. (Since immutable types cannot be altered, immutable parts of the structure may not be copied.) Such an operation can be performed with the `deepcopy` method of the `copy` module,

```
>>>import copy
>>>list2 = copy.deepcopy(list1)
```

The result of this form of copying is shown here:



Thus, the reassignment of any part (top level or sublist) of one list will not result in a change in the other. It is up to you as the programmer to determine which form of copy is needed for lists, and other mutable types.

Your Turn

From the Python Shell, enter the following and observe the results.

```
>>> import copy


>>> list1 = [10, 20, 30, 40]              >>> list1 = [10, 20, 30, [40]]
>>> list2 = list1                         >>> list2 = copy.deepcopy (list1)
>>> id(list1) == idList2)                 >>> id(list1) == idList2)
???                                       ???


>>> list1[0]=60                           >>> list1[0]=60
>>> list1                                 >>> list1
???                                       ???


>>> list2                                 >>> list2
???                                       ???


>>> list1 = [10, 20, 30, [40]]            >>> list1 = [10, 20, 30, (40)]
>>> list2 = list1                         >>> list2 = copy.deepcopy (list1)
>>> id(list1) == idList2)
???


>>> list1[0]=60
>>> list1[3][0] = 90
>>> list1
???


>>> list2
???
```

```
>>> list1[3][0] = 90
???


>>> list1[3][0] = (100,)
>>> list1
???



>>> list2
???
```

Concepts and Procedures

1. All objects have a set of ____________________ and ___________________.

2. The ___________________ operator is used to select members of a given object.

3. Functions that are part of an object are called_______________.

4. There are two values associated with every object in Python, the
value and the _________________________ value.

5. When memory locations are *deallocated*, it means that,

a) The memory locations are marked as unusable for the rest of the program execution.
b) The memory locations are marked as available for reuse during the remaining program execution.

6. Garbage collection is the process of automatically identifying which areas of memory can be deallocated. (TRUE/FALSE)

7. Indicate which of the following is true, and explain why.

a) When one variable is assigned to another holding an integer value, if the second variable is assigned a new value, the value of the first variable will change as well.

b) When one variable is assigned to another holding a list of integer values, if the second variable assigns a new integer value to an element in the list, the list that the first variable is assigned to will be changed as well.

Problem Solving

1. Indicate exactly what the contents of `lst1` and `lst2` would be after each of the following set of assignments,

```
(a) lst1 = [10, 20, 30]    (b) lst1 = [10, 20, 30]    (c) lst1 = [10, 20, 30]
    lst2 = [10, 20, 30]        lst2 = lst1                lst2 = list(lst1)
    lst1[2] = 50               lst1[2] = 50               lst1[2] = 50
```

2. Indicate which of the following set of assignments would result in automatic garbage collection in Python.

```
(a) lst1 = [1, 2, 3]    (b) str1 = 'Hello World'    (c) tuple1 = (1, 2, 3)
    lst2 = [5, 6, 7]        str2 = 'Nice Day'           tuple2 = tuple1
    lst1 = lst2             str3 = str1                 tuple1 = (4, 5, 6)
```

3. For the set of assignments in question 1, indicate how both the id method and is operator can be used to determine if lists `lst1` and `lst2` are each referencing the same list instance in memory.