PES UNIVERSITY

Department of Computer Science and Engineering

PES University, Bangalore, India

Python For Computational Problem Solving (UE22CS151A)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

Python Collections (data-structures / compound data-types)

Note: A data structure is a specialized format for organizing, processing, retrieving and storing data. Data structures are designed to arrange data to suit a specific purpose.

There are four collection data types in the Python programming language:

- **List** is a collection which is **ordered** and **mutable**. Allows duplicate members.
- **Tuple** is a collection which is **ordered** and **immutable**. Allows duplicate members.
- **Set** is a collection which is **unordered**, **unindexed** and **mutable**. **No duplicate members**.
- **Dictionary** is a collection which is **unordered** (till 3.6)/ **ordered** (from 3.7), **indexed** and **mutable**. **No duplicate keys**.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

Lists in Python - list() : Mutable Dynamic Arrays

Lists are just like dynamic sized arrays, declared in other languages.

Python's lists are implemented as dynamic arrays behind the scenes.

The way that data is organized has a significant impact on how effectively it can be used. One of the most obvious and useful ways to organize data is as a list.

Let us list out the characteristics of a list

1. It is a mutable data structure having 0 or more elements
2. There is no name for each element in the list
3. The elements are ordered and accessed by using index or subscript
4. The list index starts from 0
5. Lists are dynamic, i.e., the size of the list is not fixed. The list can grow or shrink. We can find the number of elements in a list at any point in time.

6. The list can be heterogeneous. The elements of the list need not be of the same type. Lists can even contain complex objects, like functions, classes, and modules.
7. Python lists can hold arbitrary elements—everything is an object in Python, including functions.
8. The type list supports a number of builtin functions for playing with the list
9. This is for those who know 'C'. It is not a linked list

List items are surrounded by square brackets and the elements in the list are separated by commas.

List objects needn't be unique. A given object can appear in a list multiple times.

Note: A list with a single object is sometimes referred to as a singleton list.

In [19]:
```
print(['apple', 'orange', 'kiwi', 'banana'][3])

fruits = ['apple', 'orange', 'kiwi', 'banana', 'cherry','jackfruit']
print(fruits)
```

```
banana
['apple', 'orange', 'kiwi', 'banana', 'cherry', 'jackfruit']
```

A list element can be any Python object.

Lists can even contain complex objects, like functions, classes, and modules

In [7]:
```
l1=[20, 3.14, 'Hello', (1,2,3), {4,5,6}, ['india','Nepal']]
print(l1)

#https://pythontutor.com/
```

```
[20, 3.14, 'Hello', (1, 2, 3), {4, 5, 6}, ['india', 'Nepal']]
```

In [20]:
```
def add(x,y):
    return x+y

import math

l2 = [int, len, add, math]
print(l2)
```

```
[<class 'int'>, <built-in function len>, <function add at 0x0000025003345F70>, <module
'math' (built-in)>]
```

Lists Are Ordered

A list is an ordered collection of objects. The order in which you specify the elements when you define a list is an innate characteristic of that list and is maintained for that list's lifetime.

Lists that have the same elements in a different order are not the same:

In [3]:
```
a = ['rama','bharata','lakshmana','shatrughna']
b = ['shatrughna','lakshmana','bharata','rama']
c = a
print(a == b)  # compares elements
print(a is b)  # The is keyword is used to test if two variables refer to the same objec
print(a is c)

print([1, 2, 3, 4] == [1, 2, 3, 4])
print([1, 2, 3, 4] == [4, 2, 3, 1])
```

```
False
False
True
```

```
True
False
```

In [4]:
```
a = ['rama','bharata','lakshmana','shatrughna']
b = ['shatrughna','lakshmana','bharata','rama']
print(a[0] is b[3])
print(id(a[0]),id(b[3]))
```

```
True
2542645677936 2542645677936
```

A list can be empty.

In [4]:
```
lst = []
print(lst)
print(len(lst))
```

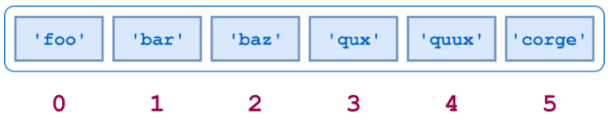
```
[]
0
```

The list elements are accessed by using index or subscript.

The list index starts from 0.

a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']

The indices for the elements in a are shown below:

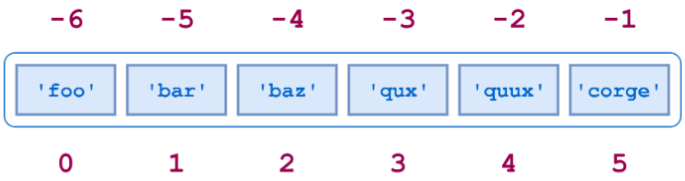


List Indices

In [3]:
```
a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
i=0
while i<6:
    print("a[{}]={}".format(i,a[i]))
    i=i+1
```

```
a[0]=foo
a[1]=bar
a[2]=baz
a[3]=qux
a[4]=quux
a[5]=corge
```

A negative list index counts from the end of the list.



Negative List Indexing

List is mutable

Lists are mutable, as list can grow or shrink. we can change an element of a list.

Once a list has been created, elements can be added, deleted, shifted, and moved around at will. Python provides a wide range of ways to modify lists.

In [25]:
```
# The format() method formats the specified value(s) and
# insert them inside the string's placeholder.
# The placeholder is defined using curly brackets

# Add value 5 to all the elements of list
l=[5,6,7,8,9]
i=0
while i<5:
    l[i]=l[i]+5
    print("l[{}]={}".format(i,l[i]))
    i=i+1
```

```
l[0]=10
l[1]=11
l[2]=12
l[3]=13
l[4]=14
```

In [3]:
```
numbers=[100,200,300,400]
print(numbers)

numbers[0]=10    # index operation is used.
#numbers[4]=500 # IndexError: list assignment index out of range
print(numbers)
```

```
[100, 200, 300, 400]
[10, 200, 300, 400]
```

List is iterable.

An iterable is any Python object capable of returning its members one at a time, permitting it to be iterated over in a for-loop.

In [11]:
```
numbers=[10,20,30,40]
for i in numbers:
    print(i,end=' ')

print()
for i in range(len(numbers)):
    print(numbers[i],end=' ')
```

```
10 20 30 40
10 20 30 40
```

List can be nested to arbitrary depth.

You have seen that an element in a list can be any sort of object. That includes another list. A list can contain sublists, which in turn can contain sublists themselves, and so on to arbitrary depth.

In [11]:
```
nested_list=[ [ [ [ [1,2],3,4],5,6],7,8],9,10]
print(nested_list)
```

```
[[[[[1, 2], 3, 4], 5, 6], 7, 8], 9, 10]
```

In [18]:
```
matrix=[[1,2,3],[4,5,6],[7,8,9]]
print(matrix)

for row in matrix:
    for num in row:
        print(num, end=' ')
    print()
print('-----')
for i in range(len(matrix)):
    for j in range(len(matrix[i])):
        print(matrix[i][j], end=' ')
    print()
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
1 2 3
4 5 6
7 8 9
-----
1 2 3
4 5 6
7 8 9
```

Several Python operators and built-in functions can also be used with lists.

In [14]:
```
# The in keyword is used to check if a value is present in a sequence.
# in - membership operator
l=[95,88,76,99,82]
print(95 in l)
print(100 not in l)
```

```
True
True
```

In [20]:
```
# There are a number of functions built into Python that
#  take lists as parameters.
l=[95,88,76,99,82]
print(len(l))
print(max(l))
print(min(l))
print(sum(l))
print(sorted(l))
print(type(l))
```

```
5
99
76
440
[76, 82, 88, 95, 99]
<class 'list'>
```

In [10]:
```
fruits = ['apple', 'orange']
print(id(fruits),fruits)
fruits = fruits + ['banana', 'kiwi']
print(id(fruits),fruits)
```

```
2542635413056 ['apple', 'orange']
2542674540800 ['apple', 'orange', 'banana', 'kiwi']
```

In [8]:
```
fruits = ['apple', 'orange', 'banana', 'kiwi']
print(fruits * 2)
```

```
['apple', 'orange', 'banana', 'kiwi', 'apple', 'orange', 'banana', 'kiwi']
```

By the way, in each example above, the list is always assigned to a variable before an operation is performed on it. But you can operate on a list literal as well:

In [17]:
```
print(['apple', 'orange', 'banana', 'kiwi'][2])
print(['apple', 'orange', 'banana', 'kiwi'][-1])
```

```
banana
kiwi
```

Assignment of one list to another causes both to refer to the same list.

In [22]:
```
l1=[1,2,3,4]      # demonstrate using pythontutor
l2=l1
#l2[0]=10
print(id(l1),l1)
print(id(l2),l2)
```

```
print(l1==l2)     # compares list contents
print(l1 is l2)   # compares list ids (ie., references)
```

```
2290206556928 [10, 2, 3, 4]
2290206556928 [10, 2, 3, 4]
True
True
```

In [25]:
```
l1=[1,2,3,400]       # demonstrate using pythontutor
l3=[1,2,3,400]
print(id(l1),l1)
print(id(l3),l3)
print(l1==l3)
print(l1 is l3)
print(id(l1[0]))
print(id(l3[0]))
```

```
2290194785920 [1, 2, 3, 400]
2290206558272 [1, 2, 3, 400]
True
False
2290089421104
2290089421104
```

In [26]:
```
print("id(l1[0]) ",id(l1[0]))
print("id(l3[0]) ",id(l3[0]))
print("id(l1[3]) ",id(l1[3]))
print("id(l3[3]) ",id(l3[3]))
```

```
id(l1[0])  2290089421104
id(l3[0])  2290089421104
id(l1[3])  2290206646672
id(l3[3])  2290206646512
```

Python range() Function

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

range() constructor has two forms of definition:

- range(stop)
- range(start, stop[, step])

range() Parameters

- start - integer starting from which the sequence of integers is to be returned
- stop - integer before which the sequence of integers is to be returned. The range of integers ends at stop - 1.
- step (Optional) - integer value which determines the increment between each integer in the sequence

In [31]:
```
print(range(10))
```

```
range(0, 10)
```

In [6]:
```
l=range(10)
print(l)
print(l[0], l[1], l[2],  l[3],  l[4],  l[5],  l[6],  l[7],  l[8],  l[9])
#print(l[10]) #IndexError: range object index out of range
```

```
range(0, 10)
0 1 2 3 4 5 6 7 8 9
```

In [1]:
```
print(list(range(10)))
print(list(range(5,10)))
print(list(range(1,10,2)))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[5, 6, 7, 8, 9]
[1, 3, 5, 7, 9]
```

Lazy evaluation

In programming language theory, lazy evaluation, or call-by-need,is an evaluation strategy which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations (sharing).

The sharing can reduce the running time of certain functions by an exponential factor over other non-strict evaluation strategies, such as call-by-name, which repeatedly evaluate the same function, blindly, regardless of whether the function can be memoized.

The benefits of lazy evaluation include:

- The ability to define control flow (structures) as abstractions instead of primitives.
- The ability to define potentially infinite data structures. This allows for more straightforward implementation of some algorithms.
- Performance increases by avoiding needless calculations, and avoiding error conditions when evaluating compound expressions.

Lazy Evaluation using Python

The range method in Python follows the concept of Lazy Evaluation. It saves the execution time for larger ranges and we never require all the values at a time, so it saves memory consumption as well. Take a look at the following example.

In Python 2, range() returns a list of integers:

for example, range(0,10) returns the list [0,1,2,3,4,5,6,7,8,9].

In Python 3, the range function returns an object of type range, which can be iterated over to yield the same sequence of integers, e.g.,

In [5]:
```
r = range(10) #range() returns an object of type range
print(list(r))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [18]:
```
for i in range(10):
    print(i, end=' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

In [12]:
```
# do this only if you need the entire list
list_of_integers = list(range(1,10,1))
print(list_of_integers)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List functions

Python offers the following list functions:

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

It is also possible to use the list() constructor to make a new list.

Using the list() constructor to make a List:

In [30]:
```
fruitlist = list(("apple", "banana", "cherry")) # note the double round-brackets
print(fruitlist)
list1 = list('Hello')
print(list1)
```

```
['apple', 'banana', 'cherry']
['H', 'e', 'l', 'l', 'o']
```

append(obj) - Appends an object to a list.

a.append(obj) appends object obj to the end of list a:

In [39]:
```
a = [11, 22, 33, 44, 55]
# Add an element at the end
a.append(66)
print(a)
```

```
[11, 22, 33, 44, 55, 66]
```

extend(iterable) - extends a list with the objects from an iterable.

extend() adds to the end of a list, but the argument is expected to be an iterable. The items in iterable are added individually:

In [11]:
```
a = [11, 22, 33, 44, 55]
#a.extend(66) # Error - 'int' object is not iterable
a.extend([66,77])
print(a)
a.extend('hello')
print(a)

b = [11, 22, 33, 44, 55]
```

```
c = (66, 77)
b.extend(c)
print(b)
```

```
[11, 22, 33, 44, 55, 66, 77]
[11, 22, 33, 44, 55, 66, 77, 'h', 'e', 'l', 'l', 'o']
[11, 22, 33, 44, 55, 66, 77]
```

In other words, extend() behaves like the + operator. More precisely, since it modifies the list in place, it behaves like the += operator.

In [1]:
```
matrix = [
         [9, 3, 8, 3],
         [4, 5, 2, 8],
         [6, 4, 3, 1],
         [1, 0, 4, 5],
         ]
```

The matrix variable holds a Python list that contains four nested lists. Each nested list represents a row in the matrix. The rows store four items or numbers each. Now say that you want to turn this matrix into the following list:

[9, 3, 8, 3, 4, 5, 2, 8, 6, 4, 3, 1, 1, 0, 4, 5]

How do you manage to flatten your matrix and get a one-dimensional list like the one above?

In [2]:
```
def flatten_extend(matrix):
    flat_list = []
    for row in matrix:
        flat_list.extend(row)
    return flat_list

flatten_extend(matrix)
```

Out[2]:
```
[9, 3, 8, 3, 4, 5, 2, 8, 6, 4, 3, 1, 1, 0, 4, 5]
```

In [3]:
```
def flatten_extend(matrix):
    flat_list = []
    for row in matrix:
        flat_list += row
    return flat_list

flatten_extend(matrix)
```

Out[3]:
```
[9, 3, 8, 3, 4, 5, 2, 8, 6, 4, 3, 1, 1, 0, 4, 5]
```

insert(index, obj) - Inserts an object into a list at a specified index.

a.insert(index, obj) inserts object obj into list a at the specified index.

Following the method call, a[index] is obj, and the remaining list elements are pushed to the right:

In [34]:
```
a = [11, 22, 33, 44, 55]

#Add at a particular position
a.insert(10,14)
#a.insert(4,40)
#a.insert(5,66)
#a.insert(6,77)
print(a)
```

```
[11, 22, 33, 44, 55, 14]
```

In [22]:
```
a = [11, 22, 33, 44, 55]

#Add at a particular position
a.insert(1,14)
a.insert(5,66)
a.insert(6,77)
print(a)
a.insert(10,77)
a.insert(-10,5)
a.insert(-1,1)
a.insert(-2,2)
print(a)
```

```
[11, 14, 22, 33, 44, 66, 77, 55]
[5, 11, 14, 22, 33, 44, 66, 77, 55, 2, 1, 77]
```

pop(index=-1) - removes an element from a list.

If the optional index parameter is specified, the item at that index is removed and returned. index may be negative.

pop() - without an argument simply removes the last item in the list.

In [24]:
```
a = [11, 22, 33, 44, 55]

# Remove at the end
x=a.pop()    # a.pop(index=-1)
print(a)
print(x)

# Remove at a particular index
x=a.pop(1)
print(a)
print(x)
#x=a.pop(5) #IndexError: pop index out of range
```

```
[11, 22, 33, 44]
55
[11, 33, 44]
22
```

This method differs from .remove() in two ways:

- You specify the index of the item to remove, rather than the object itself.
- The method returns a value: the item that was removed.

remove(obj) - removes an object from a list.

a.remove(obj) removes object obj from list a. If obj isn't in a, an exception is raised:

In [53]:
```
a = [11, 22, 33, 44, 55]

# Remove based on the value-remove() does not return any value.
a.remove(33)   # returns None
print(a)
```

```
[11, 22, 44, 55]
```

Check whether an element exists in list

In [25]:
```
a = [11, 22, 33, 44, 55]

# Check whether an element exists
```

```
print(55 in a)
print(88 not in a)
```

```
True
True
```

index(obj)

This method returns the index of the first element with the specified object. The index() method returns the index of the first matched item.

In [1]:
```
a = [11, 22, 33, 44, 55, 78, 44]
# Find the position of the element.
print(a.index(44))
```

```
3
```

count(obj)

This method returns the count of the number of occurrences of the object in the list.

In [46]:
```
# Count the number of occurrences of an element
b = [11, 22, 11, 33, 11]
print(b.count(11))
```

```
3
```

sort(key=None, reverse=False)

sort() method sorts the list items in ascending order. An optional function can be used as a key. The optional reverse flag allows to reverse to descending order.

In [36]:
```
# Arrange the elements in order
c1 = [33, 11, 55, 44, 22, 25]
c1.sort()
print(c1)
c1.sort(reverse=True)
print(c1)
```

```
[11, 22, 25, 33, 44, 55]
[55, 44, 33, 25, 22, 11]
```

Python builtin method sorted() to sort a list

In [28]:
```
numbers = [6, 9, 1, 3]
#print(numbers)
print(sorted(numbers))  # provides an ordered list as a return value
print(numbers)
print(sorted(numbers,reverse=True))
```

```
[1, 3, 6, 9]
[6, 9, 1, 3]
[9, 6, 3, 1]
```

The output from this code is a new, sorted list. When the original variable is printed, the initial values are unchanged.

The above example shows four important characteristics of sorted():

1. The function sorted() did not have to be defined. It's a built-in function that is available in a standard installation of Python.
2. sorted(), with no additional arguments or parameters, is ordering the values in numbers in an ascending order, meaning smallest to largest.

3. The original numbers variable is unchanged because sorted() provides sorted output and does not change the original value in place.
4. When sorted() is called, it provides an ordered list as a return value.

This last point means that sorted() can be used on a list, and the output can immediately be assigned to a variable:

In [6]:
```
numbers = [6, 9, 3, 1]
print(numbers)
sorted_numbers = sorted(numbers)
print(sorted_numbers)
```

```
[6, 9, 3, 1]
[1, 3, 6, 9]
```

Copy a List

You cannot copy a list simply by typing

list2 = list1

because list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2.

In [32]:
```
list1 = [5,6,7,8,9]
list2 = list1

list1.insert(5,10)
print(list2)
```

```
[5, 6, 7, 8, 9, 10]
```

There are ways to make a copy, one way is to use the built-in List method copy().

1. Make a copy of a list with the copy() method:

In [36]:
```
fruits = ["apple", "banana", "cherry"]
# .copy() - makes shallow copy
mylist = fruits.copy()  # only one level copy of child object references
print(mylist)           # won't create copies of the child objects themselves
```

```
['apple', 'banana', 'cherry']
```

In [37]:
```
mylist.insert(3,'papaya')
print(mylist)
print(fruits)
```

```
['apple', 'banana', 'cherry', 'papaya']
['apple', 'banana', 'cherry']
```

However, copy method won't work for custom objects and, on top of that, it only creates shallow copies. For compound objects like lists, dicts, and sets, there's an important difference between shallow and deep copying:

- A shallow copy means constructing a new collection object and then populating it with references to the child objects found in the original. In essence, a shallow copy is only one level deep. The copying process does not recurse and therefore won't create copies of the child objects themselves.
- A deep copy makes the copying process recursive. It means first constructing a new collection object and then recursively populating it with copies of the child objects found in the original. Copying an

object this way walks the whole object tree to create a fully independent clone of the original object and all of its children.

Making Shallow Copies

In the example below, we'll create a new nested list and then shallowly copy it with the list() factory function:

In [39]:
```
x1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
y1 = list(x1)  # Make a shallow copy
```

This means y1 will now be a new and independent object with the same contents as x1. You can verify this by inspecting both objects:

In [40]:
```
print(x1)
print(y1)
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In [42]:
```
x1.append('Hello')
print(x1)
print(y1)
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9], 'Hello']
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In [43]:
```
print(id(x1[0]))
print(id(y1[0]))
```

```
2542674714176
2542674714176
```

However, because we only created a shallow copy of the original list, y1 still contains references to the original child objects stored in x1.

These children were not copied. They were merely referenced again in the copied list.

Therefore, when you modify one of the child objects in x1, this modification will be reflected in y1 as well—that's because both lists share the same child objects. The copy is only a shallow, one level deep copy:

In [45]:
```
x1[0][0] = 55
print(x1)
print(y1)
```

```
[[55, 2, 3], [4, 5, 6], [7, 8, 9], 'Hello']
[[55, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In the above example we (seemingly) only made a change to x1. But it turns out that both sublists at index 0 in x1 and y1 were modified. Again, this happened because we had only created a shallow copy of the original list.

Had we created a deep copy of x1 in the first step, both objects would've been fully independent.

This is the practical difference between shallow and deep copies of objects.

Now you know how to create shallow copies of some of the built-in collection classes, and you know the difference between shallow and deep copying.

The questions we still want answers for are:

- How can you create deep copies of built-in collections?

- How can you create copies (shallow and deep) of arbitrary objects, including custom classes?

The answer to these questions lies in the copy module in the Python standard library.

copy module provides a simple interface for creating shallow and deep copies of arbitrary Python objects.

In [38]:
```
# importing "copy" for copy operations
import copy
x1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
x2 = copy.deepcopy(x1)  # Make a deepcopy copy
print(x1)
print(x2)
print('-------------------------------')
x1[0][0] = 55
print(x1)
print(x2)
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
-------------------------------
[[55, 2, 3], [4, 5, 6], [7, 8, 9]]
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In [40]:
```
import copy
x1 = [[1, 2, 'Hi'], [4, 5, 6], [7, 8, 9]]
x2 = copy.deepcopy(x1)
print(id(x1[0][2]),id(x2[0][2])) # same because string is immutable object
```

```
2561898235504 2561898235504
```

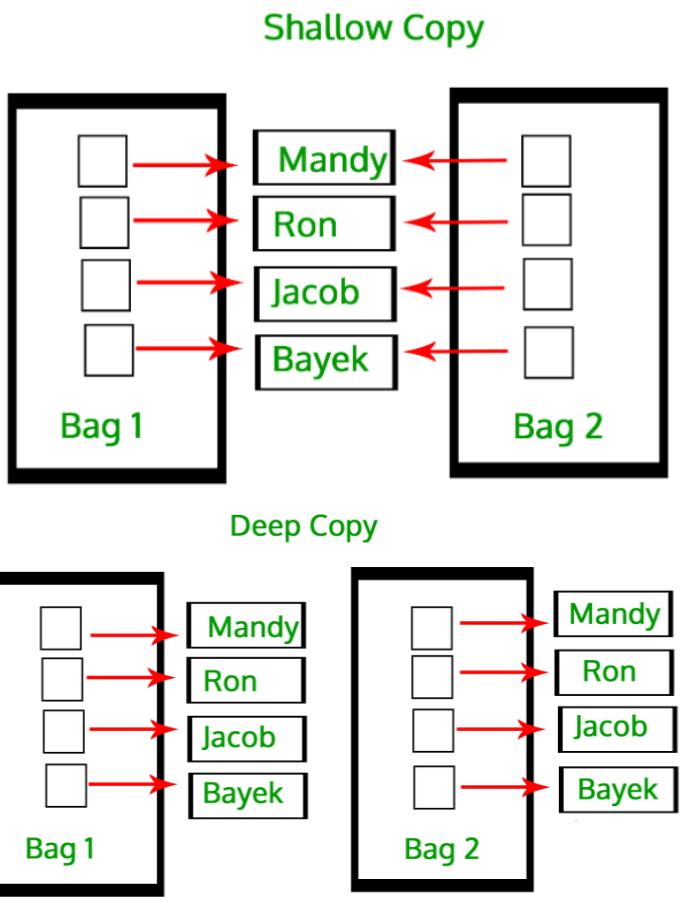
Note: Shallow copy

- Shallow copy constructs a new collection object and then populates it with references to the child objects found in the original.
- The copying process does not recurse and therefore won't create copies of the child objects themselves.
- In the case of shallow copy, a reference of an object is copied into another object. It means that any changes made to a copy of an object do reflect in the original object.
- In python, this is implemented using the "copy()" function.

Important Points:

The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances):

- A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.
- A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.



2. Make a copy of a list with the list() constructor:

The list() constructor returns a list in Python.

syntax: list([iterable or sequence])

In [30]:
```
fruitlist = ["apple", "banana", "cherry"]
mylist = list(fruitlist)
print(mylist)
```

```
['apple', 'banana', 'cherry']
```

Create lists from string and tuple

In [1]:
```
# creating empty list with the list() constructor
lst=list()
print(lst)
```

```
[]
```

In [6]:
```
vowel_string = 'aeiouAEIOU'    # vowel string

lst1=list(vowel_string)
print(lst1)

lst2=list('BCDFGJKLMNPQSTVXZHRWY')  # consonant string
print(lst2)
```

```
['a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U']
['B', 'C', 'D', 'F', 'G', 'J', 'K', 'L', 'M', 'N', 'P', 'Q', 'S', 'T', 'V', 'X', 'Z',
'H', 'R', 'W', 'Y']
```

In [7]:
```
vowel_tuple = ('a', 'e', 'i', 'o', 'u') # vowel tuple
lst3=list(vowel_tuple)
print(lst3)
```

```
['a', 'e', 'i', 'o', 'u']
```

Create lists from set and dictionary

In [33]:
```
# vowel set
vowel_set = {'a', 'e', 'i', 'o', 'u'}
lst4=list(vowel_set)
print(lst4)
print(lst4)
print(lst4)
```

```
['e', 'i', 'u', 'o', 'a']
['e', 'i', 'u', 'o', 'a']
['e', 'i', 'u', 'o', 'a']
```

In [30]:
```
# vowel dictionary
vowel_dictionary = {'a': 1, 'e': 2, 'i': 3, 'o':4, 'u':5}
l5=list(vowel_dictionary)
print(l5)
```

```
['a', 'e', 'i', 'o', 'u']
```

Joining Two Lists

There are several ways to join, or concatenate, two or more lists in Python.

1.One of the easiest ways are by using the + operator.

In [19]:
```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list3 = list1 + list2
print(list3)
```

```
['a', 'b', 'c', 1, 2, 3]
```

2.Another way to join two lists are by appending all the items from list2 into list1, one by one:

In [18]:
```
# Append list2 into list1:
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

for x in list2:
    list1.append(x)

print(list1)
```

```
['a', 'b', 'c', 1, 2, 3]
```

3.Another way to join two lists are by appending all the items from list2 into list1, at once:

In [20]:
```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list1.extend(list2)
print(list1)
```

```
['a', 'b', 'c', 1, 2, 3]
```

Slicing in lists

Slicing is a flexible tool to build new lists out of an existing list. Python supports slice notation for any sequential data type like lists, strings and tuples.

After getting the list, we can get a part of it using python's slicing operator which has following syntax :

list_name[start : stop : steps]

If steps is +ve value, traversal is left to right.

- slicing will start from index start
- will go up to stop(but not including the stop value) in step of steps.
- Default value of start is 0, stop is length of list and for steps it is 1

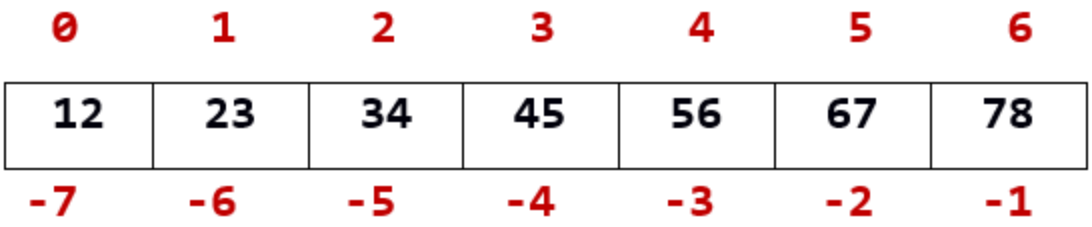
If steps is -ve value, traversal is right to left.

- slicing will start from index start
- will go up to stop(but not including the stop value) in step of steps.
- Default value of start is -1, stop is -(length of list+1)

Note:

- We create a sublist of a list by specifying a range of indices.
- Semantically specifying the slice is similar to that of range function. But syntactically, it is different.
- The result of slicing is a new list.

```
b = [12, 23, 34, 45, 56, 67, 78]
```



In [19]:
```
b = [12, 23, 34, 45, 56, 67, 78]
print(b[2:5])
print(b[:5])
print(b[2:])
print(b[2:6:2])
print(b[::2])
print(b[::-1])
```

```
[34, 45, 56]
[12, 23, 34, 45, 56]
[34, 45, 56, 67, 78]
[34, 56]
[12, 34, 56, 78]
[78, 67, 56, 45, 34, 23, 12]
```

In [4]:
```
b = [12, 23, 34, 45, 56, 67, 78]
print(b[2:5])    # [34, 45, 56]
print(b[:5])     # b[0:5] # [12, 23, 34, 45, 56]
print(b[2:])     # b[2:len(b)] # [34, 45, 56, 67, 78]
print(b[2:6:2]) # init : 2, final value one past the end : 6; step 2 # [34, 56]
print(b[::2])    # init : 0, final value one past the end : len(list); step : 2
                 # [12, 34, 56, 78]
print(b[::-1])   # reverse the elements of the list # [78, 67, 56, 45, 34, 23, 12]
```

```
[34, 45, 56]
```

```
[12, 23, 34, 45, 56]
[34, 45, 56, 67, 78]
[34, 56]
[12, 34, 56, 78]
[78, 67, 56, 45, 34, 23, 12]
```

Note: Negative value of steps shows right to left traversal instead of left to right traversal that is why [: : -1] prints list in reverse order.

Taking n first elements of a list

Slice notation allows you to skip any element of the full syntax. If we skip the start number then it starts from 0 index:

In [2]:
```
nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
print(nums[:6])
```

```
[10, 20, 30, 40, 50, 60]
```

Leaving n first elements of a list

In [3]:
```
nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
print(nums[6:])
```

```
[70, 80, 90]
```

Taking n last elements of a list

Negative indexes allow us to easily take n-last elements of a list:

In [5]:
```
nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
print(nums[-3:])  # last 3 elements, start from index -3 go till -1
```

```
[70, 80, 90]
```

Here, the stop parameter is skipped. That means you take from the start position, till the end of the list. We start from the third element from the end (value 70 with index -3) and take everything to the end.

Leaving n last elements of a list

In [35]:
```
nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
print(nums[:-3])  # leaving last 3 elements, start from index 0 go till -4
```

```
[10, 20, 30, 40, 50, 60]
```

Leaving n first and n last elements of a list

In [15]:
```
nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
print(nums[2:-2])
```

```
[30, 40, 50, 60, 70]
```

We can freely mix negative and positive indexes in start and stop positions:

In [20]:
```
nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
print(nums[-1:8])
print(nums[-1:9])
print(nums[-1:10])
print(nums[-1:7])
print(nums[-1:-7])
print(nums[-1:6:-1])
print(nums[-5:-1])
```

```
[]
[90]
```

```
[90]
[]
[]
[90, 80]
[50, 60, 70, 80]
```

Note:

If a is a list, a[:] returns a new object that is a copy of a:

In [13]:
```
a = ['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
b = a[:]
print(b)
print(b is a)
```

```
['foo', 'bar', 'baz', 'qux', 'quux', 'corge']
False
```

Exercise Problems:

1.Write a program to find the difference between successive elements of a list.

- Given: fallofwickets_scoreslist = [10, 50, 100, 145, 150, 175]
- Find all wickets partnership scores

In [3]:
```
fallofwickets_scoreslist = [10, 50, 100, 145, 150, 175]
len_scoreslist = len(fallofwickets_scoreslist)

partnership_scorelist = []
partnership_scorelist.append(fallofwickets_scoreslist[0])
for i in range(len_scoreslist - 1) :
    partnership_scorelist.append(fallofwickets_scoreslist[i + 1] - fallofwickets_scoresl


print("Fall of wickets at scores :",end='')
for score in fallofwickets_scoreslist:
    print(score, end = " ")
print()

print("Partnership Scores: ")
i=1
for score in partnership_scorelist:
    print(i,'wicket partnership score =',score)
    i=i+1
```

```
Fall of wickets at scores :10 50 100 145 150 175
Partnership Scores:
1 wicket partnership score = 10
2 wicket partnership score = 40
3 wicket partnership score = 50
4 wicket partnership score = 45
5 wicket partnership score = 5
6 wicket partnership score = 25
```

2.Runs_scored_Teams = [288,176,190,333,254,177,281,290,320]

- Find the highest runs scored by a team.
- Find the Second highest runs scored by a team.
- Check for same scores
- Remove the lowest runs from the list

In [ ]:
```
Runs_scored_Teams = [288,176,190,333,254,177,281,290,320]
```

```
Runs_scored_Teams.sort()
print(Runs_scored_Teams)
print('Highest runs =',Runs_scored_Teams[-1])
print('Second highest runs =',Runs_scored_Teams[-2])
print('Unique scores are present ',len(Runs_scored_Teams) == len(set(Runs_scored_Teams)))
print('Removed Lowest runs =',Runs_scored_Teams.pop(0))

print(Runs_scored_Teams)
```

3.Given a list: Write python statements for the following:

- add 'Ravindra Jadeja' and 'R. Ashwin' to the list together
- delete " KL Rahul" from the list
- add " Shreyas Iyer" after Mayank Agarwal
- count the list players

In [4]:
```
test_team=['Ajinkya Rahane', 'KL Rahul', 'Mayank Agarwal', 'Cheteshwar Pujara', 'Shubman

test_team.extend(['Ravindra Jadeja', 'R. Ashwin'])
test_team.remove("KL Rahul")
test_team.insert(2,"Shreyas Iyer")
print(len(test_team))
```

```
8
```

4.Swap first and last elements of a list using only a single varaible

In [7]:
```
lst=[1,2,3,4,5,6]
print(lst)

t=lst[-1]
lst[-1]=lst[0]
lst[0]=t

print(lst)
```

```
[1, 2, 3, 4, 5, 6]
[6, 2, 3, 4, 5, 1]
```

5.Swap first and last elements of a list without using a single varaible

In [6]:
```
lst=[1,2,3,4,5,6]
print(lst)

lst[0],lst[-1] = lst[-1],lst[0]

print(lst)
```

```
[1, 2, 3, 4, 5, 6]
[6, 2, 3, 4, 5, 1]
```

6.What is the output of following problems?

In [9]:
```
l=[1,2,3,4]
print(l.append(5))
print(l)
print(l.pop())
print(l)
```

```
None
[1, 2, 3, 4, 5]
5
[1, 2, 3, 4]
```

In [18]:
```
b = [12, 23, 34, 45, 56, 67]
print(b[-2::],b[:-2:], b[::-2])
```

```
[56, 67] [12, 23, 34, 45] [67, 45, 23]
```

In [20]:
```
lst = [55,45,65,15]
print(lst.sort())
print(lst)
```

```
None
[15, 45, 55, 65]
```

In [50]:
```
b = [12, 23, 34, 45, 56, 67]
print(b[2::], b[:2:], b[::-1])
```

```
[34, 45, 56, 67] [12, 23] [67, 56, 45, 34, 23, 12]
```

In [55]:
```
cars = ['audi', 'bmw', 'ford', 'toyota']
for car in cars:
    if car == 'bmw':
        print(car.upper())
    else:
        print(car.title())
```

```
Audi
BMW
Ford
Toyota
```

In [16]:
```
l=[1,2,3,4]
for l[-1] in l:    # l[-1]=l[1], l[-1]=l[2], l[-1]=l[3], l[-1]=l[4]
    print(l)
```

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

In [5]:
```
# shallow copy example using copy module
lst1=[1,2,3,4,5,6]
import copy
lst2=copy.copy(lst1)
print('lst1 =',lst1)
print('lst2 =',lst2)
print('id(lst1) =',id(lst1))
print('id(lst2) =',id(lst2))
```

```
lst1 = [1, 2, 3, 4, 5, 6]
lst2 = [1, 2, 3, 4, 5, 6]
id(lst1) = 1936246002176
id(lst2) = 1936246003584
```

In [10]:
```
l1=[1,2,3,4]
l2=[1,2,3,4]
print(l1==l2)    # compares elements
print(l1 is l2) # compares references
```

```
True
False
```

In [1]:
```
numbers=list(map(int,input("Enter 10 numbers with space: ").split()))
print(numbers)
```

```
Enter 10 numbers with space: 1 2 3 4 5 6 7 8 9 10
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [1]:
```
l=[1,2,3]*0
print(l)
```

```
[]
```

In [4]:
```
A=[10,20,30,40,50]
print(A[0:0])
```

```
[]
```

In [1]:
```
lst=[ 12, [23,45], [([55],67,89),99] ]
print(lst[2][0][0])
```

```
[55]
```

In [8]:
```
a = [1, 2, 3, 4]
n = len(a) - 1
for i in range(len(a)):
    t=a[i]
    a[i]=a[n-i]
    a[n-i]=t
    print(a)
print(a[0], a[3])
```

```
[4, 2, 3, 1]
[4, 3, 2, 1]
[4, 2, 3, 1]
[1, 2, 3, 4]
1 4
```

In [18]:
```
for i in range(5,1,-4) :
    print(i+1, end = " ")
```

```
6
```

In [21]:
```
def f1():
    return
    return
x=f1()
print(x)
```

```
None
```

In [23]:
```
list = ['modi', 'Yogi', 'jaitley','swaraj']
for mnstr in list:
    print(mnstr[0]+mnstr[1]+mnstr[2]+mnstr[3])
```

```
modi
Yogi
jait
swar
```

References:

1. https://realpython.com/python-lists-tuples/
2. https://www.w3schools.com/python
3. https://docs.python.org/

In [24]:
```
s = "INDIA";
s.replace("I", "").lower()
print(s)
```

```
INDIA
```

In [26]:
```
def f1():
    return 1,2,3;print("in f1")
x=f1()
print(x)
```

```
(1, 2, 3)
```

In [28]:
```
s = set("abc") | set("bca") | set("cab")
print(s)
```

```
{'c', 'b', 'a'}
```

In [ ]:
```

```