Data structures :

- Lists, tuples, dictionaries

- This section will cover :
  - lists and tuples

Lists

- Lists are used to store multiple items in a single variable.
- Each value has its index (starting at zero) and can appear in the list multiple times.
- A list is defined as a comma-separated list of values, enclosed in square brackets:

```
x =[18, 20, 25, 29, 31]
```

List indexing (1)

```
x =[18, 20, 25, 29, 31]
```

- To access the list elements use indices. The first element has index zero, the second one has index 1, etc.

```
print(x[0])      #18
print(x[1])      #20
```

- Negative indices: -1 to access the last element, -2 for the
- the second-to-last element, and so on.

```
print(x[-1])    #31
print(x[-2])    #29
```

- Indexing a non-existent list element will result in an error.

```
print(x[5])
#IndexError: list index out of range
```

List indexing (2)

```
x =[18, 20, 25, 29, 31]
```

- The same indexing that we have used to read the elements of a list can be used to change them:

```
x[3] = 26
x[-1] = 33
print(x)
#[18, 20, 25, 26, 33]
```

List Elements

- In some languages the elements of a list have to be of a same type. However, in Python, the list *elements* can be any type.

```
[10, 20, 30, 40]
['apple', 'pear', 'banana']
```

  - And can be mixed type:

```
['spam', 2.0, 5, [10, 20]]
```

    - a string, a float, an integer, and **another list**
    - Lists inside lists are known as nested lists.

Lists Vs. Strings

- Both lists and strings are **sequences**, and the [ ] operator is used to access an element in any sequence
- There are two differences between lists and strings:
  - Lists can hold values of any type, whereas strings are sequences of characters
  - Strings are *immutable* - you cannot change the characters in the sequence:

```
greeting = "Hello, world!"
greeting[0] = 'J'              # ERROR!
```

  - Lists are *mutable:*

```
numbers = [42, 123]
numbers[1] = 5          # [42, 5]
```

Loop Over the Index Values

- Loop through the list items in values by referring to their index number. Use the range () and len() functions to create the iterable.

```
for i in range(len(values)) :
        print(i, values[i])
```

- Or print all items in the list, one by one:

```
for element in values :
    print(element)
```

List Operations

1. Appending / Extending
2. Finding an Element
3. pop() and remove()
4. Inserting an Element at specific position
5. Sorting
6. Copying Lists
7. Slices of a List

1. Appending / Extending

- Add a **single element** to the end of a list with append()

```
x = [18,  20,  25]
print(len(x))    #3
x.append(29)
print(x)       #[18, 20, 25, 29]
print(len(x))       #4
```

- To append more than one element to the list use extend():

```
x = [18,  20,  25]
print(x)       #[18, 20, 25]
x.extend([29,31])
print(x)       #[18, 20, 25, 29, 31]
```

2. Finding an Element

- in operator - to know whether an element is present in a list:

```
x =[18, 20, 25, 29, 31]
print("18 in x?", 18 in x)          #True
print("19 in x?", 19 in x)          #False
print("31 in x?", 31 in x)          #True
print("31 not in x?", 31 not in x) #False
```

- Use the index() method to know the position at which an element occurs . Yields the index of the **first** match

```
print("Index of 18:", x.index(18))  # 0
print("Index of 25:", x.index(25))  # 2
```

Self Check Question

- The following code would cause an error:

```
x =[18, 20, 25, 29, 31]
print("Index of 35:", x.index(35))
# ValueError: 35 is not in list
```

- What additional code could you use to ensure that an error is not displayed?

3. pop() and remove()

- pop() - removes the element at a given position (if you know the index of the element.)
- pop() - with an empty parameter will remove the last element

```
q = ['a', 'b', 'c', 'd']
q.pop(1)
print(q)          # ['a', 'c', 'd']
q.pop()
print(q)      # ['a', 'c']
```

- remove() - if you do not know the index, this will remove the first matching element.

```
q = ['a', 'b', 'c', 'd']
q.remove('b')
print(q)      # ['a', 'c', 'd']
```

3. pop() review

- What is printed by the following?

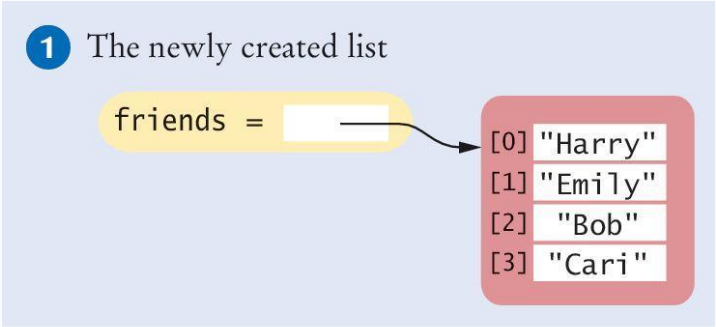
```
alist = [4, 2, 8, 6, 5]
alist.pop(2)
alist.pop()
print(alist)
```

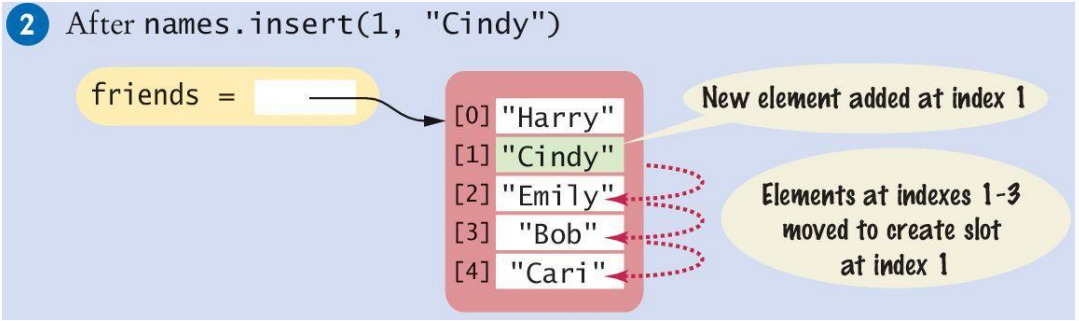
4. Insert an Element at specific position

- Sometimes the order in which elements are added to a list is important
  - A new element has to be inserted at a specific position in the list:

```
#1
friends = ["Harry", "Emily",
 "Bob", "Cari"]


#2
friends.insert(1, "Cindy")
```





5. Sorting

- The sort() method sorts a list of numbers or strings. It modifies the list.

```
values = [1, 16, 9, 4]
values.sort()                    #[1, 4 , 9, 16]
values.sort(reverse=True)    # descending
print(values)                  #[16, 9, 4, 1]
```

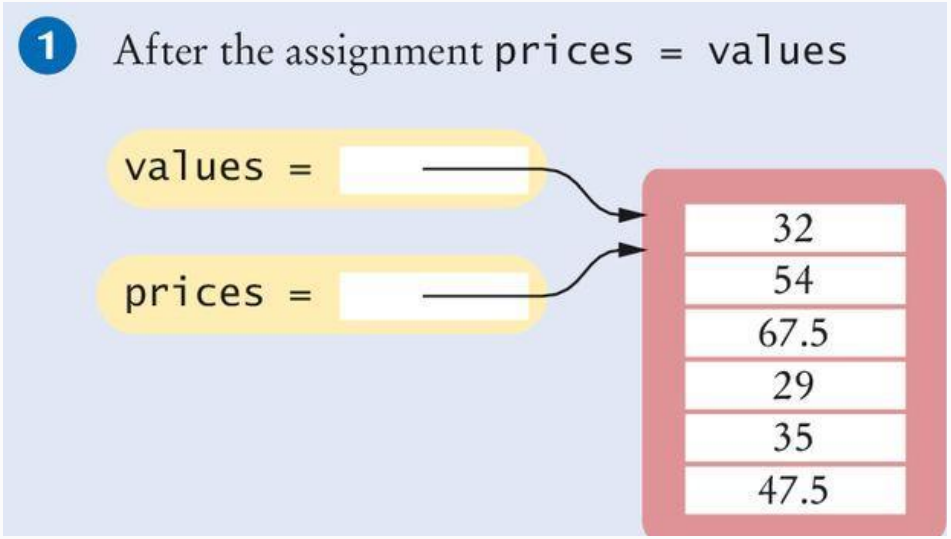
- The sorted() method creates **a new list** containing a sorted version of the list. The original list is unsorted.

```
values = [1, 16, 9, 4]
sorted(values)                    #[1, 4 , 9, 16]
print(values)                     #[1, 16, 9, 4]
```

6. Copying Lists

- List variables do not themselves hold list elements
- They hold a reference to the actual list
- If you copy the reference, you get another reference to the same list:

```
prices = values
```

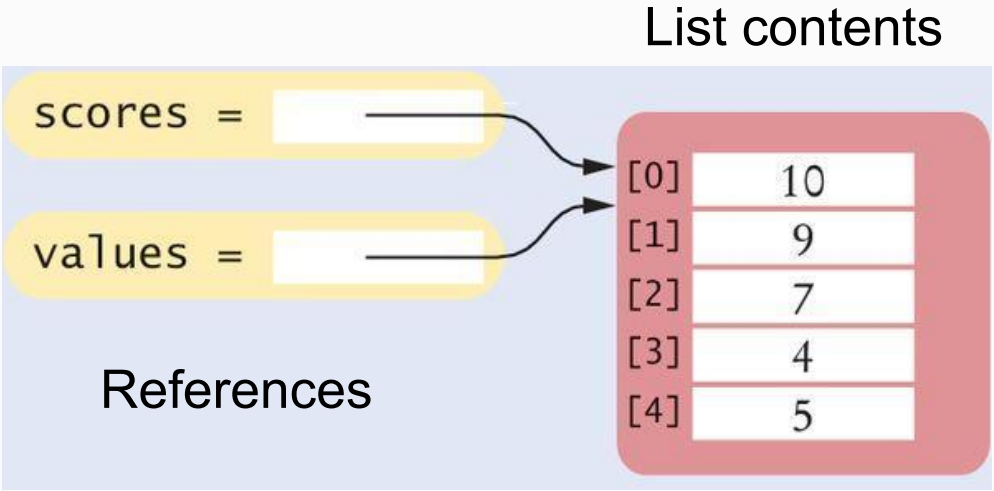


List Aliases

- When you copy a list variable into another, both variables refer to the same list
  - The second variable is an *alias* for the first because both variables reference the same list

```
scores = [10, 9, 7, 4, 5]
values = scores  #Copying list reference
```

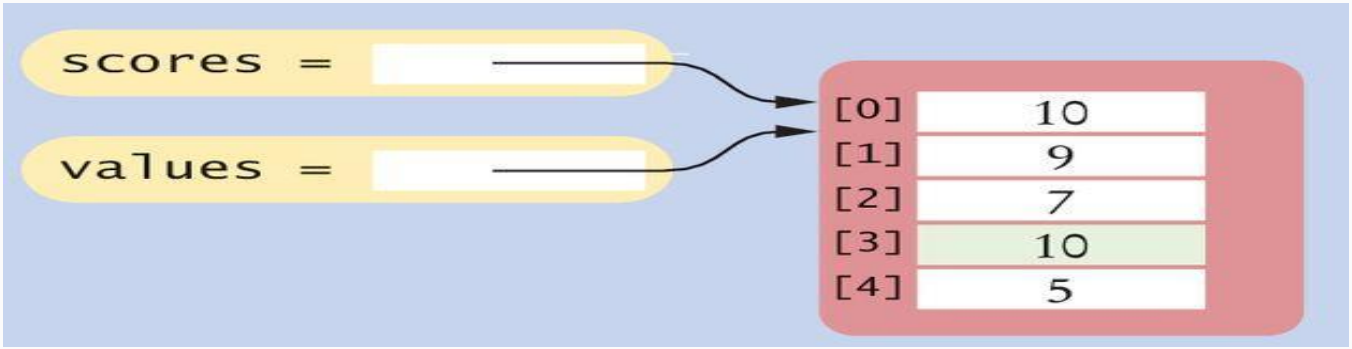
A list variable specifies the location of a list. Copying the reference yields a second reference to the same list.



Modifying Aliased Lists

- You can modify the list through either of the variables:

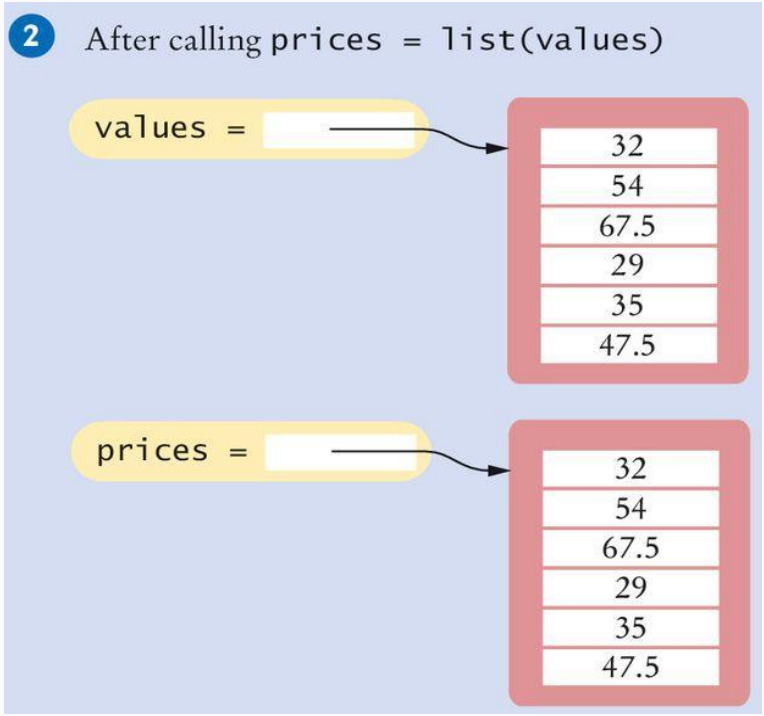
```
scores[3] = 10
print(values[3])    # Prints 10
```



Copying Lists (2)

- Sometimes, you want to make a copy of a list; that is, a new list that has the same elements in the same order as a given list
- Use the list() function:

```
prices = list(values)
```



7. Slices of a List

- A list of temperatures, one per month:

```
temp = [18, 21, 24, 33, 39, 40, 39, 36, 30, 22, 18]
```

- To obtain temperatures for the third quarter, with index values 6, 7, and 8 use the slice operator:

```
thirdQuarter = temp[6 : 9]
```

- To replace the values in elements 6, 7, and 8:

```
temp[6 : 9] = [45, 44, 40]
# [18, 21, 24, 33, 39, 40, 45, 44, 40, 22, 18]
```

- Which elements would the following slices obtain?

```
temp[ : 6]
temp[6 : ]
```

Exercises 1-6

1. Create a list "mylist" containing integers: 1, 2, 3, 4, 5
2. Print the second item in the list.
3. Use a negative index to print the second-to-last element.
4. Use list slicing to print the second to the fourth item in the list. E.g., [2, 3, 4]
5. Replace the first list item with the value 10.
6. Append the number 11 to the list.

Tuples (1)

- A tuple is similar to a list, but once created, its contents cannot be modified .
- A tuple as immutable (unchangeable) version of a list.
- A tuple is created by specifying its contents as a comma-separated sequence enclosed in parentheses:

```
triple = (5, 10, 15)
```

- If you prefer, you can omit the parentheses:

```
triple = 5, 10, 15
```

- A tuple with one value must have an ending comma:

```
single = 5,
```

Tuples (2)

- Processing a tuple in the same ways as a list:

```
a_tuple = (17, 13, 19, 23)
print(a_tuple[0])     #17
print(a_tuple[-1])    #23
print(a_tuple[1:-1]) #(13, 19)
print(sorted(a_tuple))  #[13, 17, 19, 23]
print(17 in a_tuple) # True
```

- However, tuple values are **immutable**. This means tuple items cannot be added, removed or replaced.
- This is useful if you want to store data that you want to remain fixed during a program.
- Also, they are faster type than lists for Python to process.

Tuples (3)

- Therefore, the following are not possible:

```
a_tuple[2] = 5                  # error
a_tuple.sort()                # error
a_tuple.extend([34, 54])   # error
a_tuple.insert(1, 39)    # error
a_tuple.append(400)          # error
a_tuple.pop()                 # error
```

- We can convert a tuple to a list:

```
a = list(a_tuple)
print(a)          #[17, 13, 19, 23]
```

- We can convert a list to a tuple:

```
b = tuple(a)      #(17, 13, 19, 23)
```

Summary: Lists & Tuples

- **lists** - comma-separated sequence enclosed in square brackets **[]**. **Mutable**. Some built-In list operations:
  - append() inserts a new element at the end of the list
  - index() to know the position of an element
  - pop() removes an element
  - remove() if you know the element but NOT the index.
  - insert() to insert a new element at a position in a list
  - list() function to copy lists
- **tuple** - comma-separated sequence enclosed in parentheses**()**. **Immutable** - useful if you want to store data that you want to remain fixed during the run of a program. Faster processing.