\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CSC249 DATA Structure and Algorithms**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

PYTHON COLLECTIONS

# **Objectives**

In this lesson, students will learn:

- How to create and use lists

- How to create and use tuples

- How to create and use sets

- How to create and use dictionaries

Python has a few built-in collections. In this lesson, we will learn how to create and use lists, tuples, sets and dictionaries.

# **1. Lists**

# **1.1 Creating Lists**

In Python, a list can be used to store multiple items in sequential order. We can use an assignment statement to create a list. Example:

score = [97, 85, 93, 76, 81, 88, 100, 72, 84, 75]

The statement above creates a list named score that holds 10 integers.

score[0]

score[1]

score[2]

score[3]

score[4]

score[5]

score[6]

score[7]

score[8]

score[9]

97

85

93

76

81

88

100

72

84

75

The following statement creates a list gpa to store GPA’s of 5 students:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]

The following statement creates a list dept to store names of 4 departments:

dept = [**'Psychology'**, **'History'**, **'Philosophy'**, **'English'**]

You can store data of different types in the same list. The following is an example:

student = [**"Peter Parker"**, 24, 2.87]

There are three items in the list above. The first item is the name of a student. The second item is the student’s number of credit hours completed. The third item is the student’s GPA. The name is a string. Number of credit hours is integer. GPA is floating point number.

The following statement creates an empty list list1, which has no elements:

list11 = []

Sometimes we create an empty list first and then add elements to it. You will see some examples soon.

Once a list is created, you can use the print function to display it. Examples:

score = [97, 85, 93, 76, 81, 88, 100, 72, 84, 75]  
gpa = [3.25, 2.87, 3.10, 3.56, 2.64]  
dept = [**'Psychology'**, **'History'**, **'Philosophy'**, **'English'**]  
print(score)  
print(gpa)  
print(dept)

When you pass the name of a list to the print function, the whole list will be displayed inside a pair of square brackets. The following is the output of the program:

[97, 85, 93, 76, 81, 88, 100, 72, 84, 75]

[3.25, 2.87, 3.1, 3.56, 2.64]

['Psychology', 'History', 'Philosophy', 'English']

# **1.2 Accessing and Altering List Elements**

You can retrieve a list element just like you retrieve the value of a variable. Examples:

print(score[2])  
print(gpa[1])  
print(dept[0])

The first statement displays 93, which is the third element in the list score. The second statement displays 2.87, which is the second element in the list gpa. The third statement displays Psychology, which is the first element in the list dept. When we retrieve a list element, it does not remove the element from the list. We just simply look up the value.

In addition to using positive indices, we can also use negative indices to access list elements. Examples:

print(score[-2])  
print(gpa[-1])  
print(dept[-3])

When negative indices are used, we are counting backward from the end of the list.

score[-10]

score[-9]

score[-8]

score[-7]

score[-6]

score[-5]

score[-4]

score[-3]

score[-2]

score[-1]

97

85

93

76

81

88

100

72

84

75

The index -1 means the last element in the list. The index -2 means the second-to-last element in the list. The index -3 means the third-to-last element in the list, and so on. Using this system, print(score[-2]) displays 84, while print(score[-6]) displays 81.

The index -0 is the same as 0. It references the first, not the last, element of the list. Therefore, the statement

print(score[-0])

is the same as the statement

print(score[0])

Both of them display 97, which is the first element of the list score.

The examples above retrieve list elements to display them. You can retrieve list elements for other purposes such as using them for calculations. Examples:

total\_first\_two\_scores = score[0] + score[1]  
avg\_last\_three\_gpas = (gpa[-1] + gpa[-2] + gpa[-3])/3

The first statement calculates the total of first two elements in the list score, while the second statement calculates the average of the last three elements in the list gpa.

We can use assignment statements to change elements of a list. Examples:

score[1] = 100  
gpa[-1] = 3.87  
dept[2] = **'Music'**

The first statement changes the second element of score to 100. The second statement changes the last element of gpa to 3.87. The third statement changes the third element of dept to Music.

# **1.3 Functions and Operations Applicable to Lists**

## 1.3.1 The sum, min and max functions

The sum, min and max functions return the sum of all list elements, the smallest element and the largest element, respectively. Example:

listx = [18, 7, 22, 8, 6]

sum = sum(listx)

print("The sum is:", sum)

min = min(listx)

print("The smallest element is:", min)

max = max(listx)

print("The largest element is:", max)

Output:

The sum is: 61

The smallest element is: 6

The largest element is: 22

## 1.3.2 The len function

The len function finds the length of a list. Example:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]

list\_length = len(gpa)

print("Number of elements in the list:", list\_length)

The len function in the second statement finds the length of the list gpa and stores it in the variable list\_length, which is displayed by the third statement:

Number of elements in the list: 5

Another example:

dept = [**'Psychology'**, **'History'**, **'Philosophy'**, **'English'**]

print(**"Number of elements in the list:"**, len(dept))

The len function in the second statement finds the length of the list dept, which is displayed by the print function.

Number of elements in the list: 4

## 1.3.3 The in operator

The in operator tests whether a value is in a list. Example:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]  
**if** 3.10 **in** gpa:  
 print(**"3.10 found in the list"**)  
**else**:  
 print(**"3.10 not in the list"**)

The in operator in the if statement tests whether 3.10 is an element in the list gpa or not. If 3.10 is found in the list, the condition is true. If 3.10 is not in the list, the condition is false.

We can also ask the user to enter a search target:

dept = [**'Psychology'**, **'History'**, **'Philosophy'**, **'English'**]

target = input(**"Enter search target: "**)  
**if** target **in dept**:  
 print(target, **"is found in the list"**)  
**else**:  
 print(target, **"is not in the list"**)

Sample test run:

Enter search target: History

History is found in the list

Another test run:

Enter search target: Music

Music is not in the list

## 1.3.4 The not in operator

The not in operator tests whether a test value is not in a list. Example:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]  
**if** 3.10 not **in** gpa:  
 print(**"3.10 not in the list"**)  
**else**:  
 print(**"3.10 is in the list"**)

## 1.3.5 Delete

We can use the del keyword to remove an element from a list. Example:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]  
print(**"List before delete:"**,gpa)  
**del** gpa[2]  
print(**"List after delete: "**, gpa)

The third statement in the program above deletes the element with index 2, i.e. the third element, from the list gpa. Before the delete operation, 3.10 was the third element. After delete, this element is no longer in the list. The list is shortened from 5 elements to 4 elements.

List before delete: [3.25 2.87 3.1 3.56 2.64]

List after delete: [3.25 2.87 3.56 2.64]

More examples of del:

del score[0]  
del dept[**-2**]

The first statement deletes the first element from the list score. The second statement deletes the second-to-last element from the list dept.

# **1.4 List Methods**

## 1.4.1 append

We can use append to add a new element to the end of a list. For example, the third statement in the following example adds 2.97 as the sixth element to the 5-element list gpa:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]  
print(**"List before append:"**, gpa)  
gpa.append(2.97)  
print(**"List after append: "**, gpa)

The list gpa has 5 elements originally. It has 6 elements after append adds 2.97 to the end of the list. This is shown by the print statements:

List before append: [3.25 2.87 3.1 3.56 2.64]

List after append: [3.25 2.87 3.1 3.56 2.64 2.97]

More examples of append:

score.append(65)  
dept.append(**'Math'**)

The first statement adds 65 as the eleventh element to the 10-element list score. The second statement adds math as the fifth element to the 4-element list dept.

append is a useful tool when we want to input data from the user and store them in a list.

score = []  
**for** i **in** range(3):  
 new\_score = int(input(**"Enter a score: "**))  
 score.append(new\_score)  
print(**"The list:"**, score)

The program above starts with an empty list. It uses a loop to input three scores from the user and add them to the list. Every time a score is entered, it is appended to the list immediately. The following is a sample test run:

Enter a score: 84

Enter a score: 92

Enter a score: 76

The list: [84, 92, 76]

## 1.4.2 insert

We can also use insert to insert an element at a specified position of a list. The third statement in the following example inserts 3.88 as the second element of the list gpa:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]  
print(**"List before insert:"**,gpa)  
gpa.insert(1, 3.88)  
print(**"List after insert: "**, gpa)

insert requires two pieces of information: the position in the list where the new element will be inserted, and the data to be inserted. The syntax gpa.insert(1, 3.88) tells the computer to insert 3.88 in the position with index equal to 1, which means adding 3.88 as the second element. Let’s look at the list before and after the insert statement:

List before insert: [3.25, 2.87, 3.1, 3.56, 2.64]

List after insert: [3.25, 3.88, 2.87, 3.1, 3.56, 2.64]

The first element 3.25 remains the same. After 3.88 is inserted as the second element, all other elements shift one position down. For example, 2.87 was originally the second element, now it is the third.

More example of insert:

score.insert(0, 88)  
dept(3, **'Math'**)

The first statement inserts 88 as the first element of the list score. The second statement inserts Math as the fourth element of the list dept.

## 1.4.3 clear

The clear method removes all elements from the list. Example:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]  
print(**"List before clear:"**,gpa)  
gpa.clear()  
print(**"List after clear: "**, gpa)

The third statement in the program above removes all elements from the list:

List before clear: [3.25 2.87 3.1 3.56 2.64]

List after clear: []

## 1.4.4 remove

We can use remove to remove elements. Let’s look at an example:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]  
print(**"List before remove:"**,gpa)  
gpa.remove(3.10)  
print(**"List after remove:"**, gpa)

The third statement in the program above removes the element 3.10 from the list gpa. The list is shortened from 5 elements to 4 elements.

List before remove: [3.25 2.87 3.1 3.56 2.64]

List after remove: [3.25 2.87 3.56 2.64]

If two or more elements have the same value, remove only removes the first one. Here is an example:

temp\_list = [92, 89, 92]  
print(**"List before remove:"**, temp\_list)  
temp\_list.remove(92)  
print(**"List after remove: "**, temp\_list)

The list temp\_list originally has three elements 92, 89 and 92. There are two 92’s in the list. The third statement only removes the first of them. There are two elements in temp\_list after remove:

List before remove: [92 89 92]

List after remove: [89 92]

## 1.4.5 pop

The pop method removes and returns an element. Let’s look at an example:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]

item\_popped = gpa.pop(1)

print("Item popped:", item\_popped)

print("List after pop:", gpa)

The second statement in the program above removes and returns the element with index 1, which is the second element in the list. The list is shortened from 5 elements to 4 elements.

Item popped: 2.87

List after pop: [3.25, 3.1, 3.56, 2.64]

If no argument is passed to the pop method, the default index is -1. That means the last element in the list will be removed and returned. Example:

gpa = [3.25, 2.87, 3.10, 3.56, 2.64]

item\_popped = gpa.pop()

print("Item popped:", item\_popped)

print("List after pop:", gpa)

The second statement in the program above removes and returns the element with index 1, which is the second element in the list. The list is shortened from 5 elements to 4 elements.

Item popped: 2.64

List after pop: [3.25, 2.87, 3.1, 3.56]

# **1.5 Iterating Over a List**

Sometimes we need to access every list element one after another. This can be done with a for loop in two different ways.

## 1.5.1 Assigning elements to loop variable

The first way to iterate over a list is assigning list elements to the loop variable one at a time. The following is an example:

student\_list = [**'Pete Li'**, **'Al Davis'**, **'Den White'**, **'Dave Fox'**]  
print(**"Elements in the list:"**)  
**for** student **in** student\_list:  
 print(student)

The variable student is the loop variable. When the loop iterates, the computer assigns a different list element to student systematically. The following is output generated by the print statement:

Elements in the list:

Pete Li

Al Davis

Den White

Dave Fox

Since there are 4 elements in the list, the for loop iterates 4 times and displays the elements.

Let’s look at another example.

temp\_list = [92, 89, 90, 95]  
hot\_days = 0  
**for** temp **in** temp\_list:  
 **if** temp > 90:  
 hot\_days = hot\_days + 1  
print(**"Number of days hotter than 90:"**, hot\_days)

This program counts number of days hotter than 90. A counter variable hot\_days is created and initialized to 0. A for loop is used to iterate over the temperature list. Elements of the list are assigned to the loop variable temp one at a time. An if statement inside the loop tests whether temp is greater than 90. If it is, we add 1 to the counter variable hot\_days. Output of the program:

Number of days hotter than 90: 2

## 1.5.2 Using Index to Iterate Over a List

The second way to iterate over a list is using index. The following is an example:

student\_list = [**'Pete Li'**, **'Al Davis'**, **'Den White'**, **'Dave Fox'**]  
print(**"Elements in the list:"**)  
**for** i **in** range(len(student\_list)):  
 print(student\_list[i])

The range function range(len(student\_list) generates the sequence 0, 1, 2, 3. These numbers are assigned to the loop variable i, which is used as an index in the print statement to access an element of student\_list. Output of the program:

Elements in the list:

Pete Li

Al Davis

Den White

Dave Fox

We can also use this technique to rewrite the hot days counting program:

temp\_list = [92, 89, 90, 95]  
hot\_days = 0  
**for** i **in** range(len(temp\_list)):  
 **if** temp\_list[i] > 90:  
 hot\_days = hot\_days + 1  
print(**'Number of days hotter than 90:'**, hot\_days)

The loop variable i, which is assigned a different value from the sequence 0, 1, 2, 3 in each iteration, is used as the index to test an element of temp\_list in the if statement. It generates the same output as the previous version of the program..

## 1.5.3 Using a Loop to Alter List Elements

We have seen two examples where you can use two different ways to iterate over a list. The loops in those two examples only use the list elements to do something (displaying or testing the elements), but none of the elements in the lists are altered. In this section, we are going to use a loop to change the list elements. You will see that one of the techniques introduced earlier works while the other does not.

Let’s try to use a loop to add 2 to every element in the list temp\_list. Originally the temp\_list is:

[92, 89, 90, 95]

After the loop, we want it to be:

[94, 91, 92, 97]

First, let’s use index to iterate over the list and add 2 to every element:

temp\_list = [92, 89, 90, 95]  
print(**'Before for loop:'**, temp\_list)  
**for** i **in** range(len(temp\_list)):  
 temp\_list[i] = temp\_list[i] + 2 *# change list element*print(**"After for loop:"**, temp\_list)

Output of the program:

Before for loop: [92, 89, 90, 95]

After for loop: [94, 91, 92, 97]

The output is exactly what we want. This technique works.

Next, let’s try to use the loop variable to iterate over the list.

temp\_list = [92, 89, 90, 95]  
print(**'Before for loop:'**, temp\_list)  
**for** temp **in** temp\_list:  
 temp = temp + 2 *# does not change list element*print(**"After for loop:"**, temp\_list)

Output of the program:

Before for loop: [92, 89, 90, 95]

After for loop: [92, 89, 90, 95]

This time the list remains the same after the loop. The reason is that the assignment statement inside the loop changes the loop variable but not the list element. Since the loop variable is just a copy of the list element, changing the loop variable has no effect on the list element itself. Therefore, whenever you need a loop to change loop elements, you need to use index to iterate over the list. That’s the only way that works.

# **1.6 List Concatenation**

We can combine two or more lists into a single one. The following is an example:

list1 = [3, 7, 2]  
list2 = [4, 1, 9, 5]  
list3 = list1 + list2  
print(**"List1:"**, list1)  
print(**"List2:"**, list2)  
print(**"List3:"**, list3)

The third statement in the program above creates a 7-element new list named list3. The first 3 elements are copied from list1, while the last 4 elements are copied from list2. The following is the output of the program:

List1: [3, 7, 2]

List2: [4, 1, 9, 5]

List3: [3, 7, 2, 4, 1, 9, 5]

Here is another example:

list1 = [**"Physics"**, **"Math"**, **"Biology"**]  
list2 = [**"History"**, **"Music"**]  
list3 = [**"Psychology"**]  
list4 = list1 + list2 + list3  
print(**"List1:"**, list1)  
print(**"List2:"**, list2)  
print(**"List3:"**, list3)  
print(**"List4:"**, list4)

The fourth statement in the program above creates a new list named list4, which is equal to the combination of list1, list2 and list3. The following is the output of the program:

List1: ['Physics', 'Math', 'Biology']

List2: ['History', 'Music']

List3: ['Psychology']

List4: ['Physics', 'Math', 'Biology', 'History', 'Music', 'Psychology']

# **1.7 Slicing**

We can use slicing to copy some or all elements from one list to another. We need to specify the starting index and ending index. The following is an example:

listx = [**"Amy"**, **"Bill"**, **"Carol"**, **"Dan"**, **"Eve"**]  
listy = listx[1:4]  
print(**"listy:"**, listy)

The syntax listx[1:4] returns part of listx. The number before the colon is the starting index, while the number after the colon is the ending index. listx[1:4] includes three elements: listx[1], listx[2] and listx[3]. It does not include listx[4] because the element with the ending index is always excluded. The following is the output of the program:

listy: ['Bill', 'Carol', 'Dan']

Here is another example:

listx = [10, 11, 12, 13, 14, 15, 16]  
listy = listx[2:6]  
print(**"listy:"**, listy)

The syntax listx[2:6] returns listx[2], listx[3], listx[4] and listx[5]. The element listx[6] is excluded. The following is the output of the program:

listy: [12, 13, 14, 15]

In addition to starting index and ending index, we can also add a step value. Example:

listx = [10, 11, 12, 13, 14, 15, 16]  
listy = listx[1:6:2]  
print(**"listy:"**, listy)

The syntax listx[1:6:2] returns listx[1], listx[3]and listx[5]. It excludes some elements in that range because the step value is 2, which increases the index by 2 every time we go to the next element. We start with listx[1] because the starting index is 1. Since 1 + 2 is 3, the next element included in the new list is listx[3]. Similarly, since 3 + 2 is 5, the next element included in the new list is listx[5]. The following is the output of the program:

listy: [11, 13, 15]

The following program copies listx[0], listx[1], listx[2] and listx[3]to the new list listy:

listx = [10, 11, 12, 13, 14, 15, 16]  
listy = listx[0:4]  
print(**"listy:"**, listy)

Here is the output of the program:

listy: [10, 11, 12, 13]

The following program copies listx[4], listx[5] and listx[6]to the new list listy:

listx = [10, 11, 12, 13, 14, 15, 16]  
ending\_index = len(listx)  
listy = listx[4:ending\_index]  
print(**"listy:"**, listy)

The length of listx is 7. Since we use it as the ending index, the program copies elements up to and including listx[6]. The following is the output of the program:

listy: [14, 15, 16]

We can shorten the program above by combining the second and third statements:

listx = [10, 11, 12, 13, 14, 15, 16]  
listy = listx[4:len(listx)]  
print(**"listy:"**, listy)

This shortened version will generate the same output as before.

Python syntax allows us to omit the starting index and the ending index when we copy list elements. If the starting element is omitted, it means starting from the first element of the list. The following is an example:

listx = [10, 11, 12, 13, 14, 15, 16]  
listy = listx[:4]  
print(**"listy:"**, listy)

The syntax listx[:4] is the same as listx[0:4]. Do not forget the colon. Here is the output of the program:

listy: [10, 11, 12, 13]

If the ending index is omitted, it means ending at the end of the list and including the last element. The following is an example:

listx = [10, 11, 12, 13, 14, 15, 16]  
listy = listx[4:]  
print(**"listy:"**, listy)

The syntax listx[4:] is the same as listx[4:len(listx)]. Do not forget the colon. Here is the output of the program:

listy: [14, 15, 16]

You can omit both the starting and ending indices at the same time. It just means copying every element in the list. The following is an example:

listx = [10, 11, 12, 13, 14, 15, 16]  
listy = listx[:]  
print(**"listy:"**, listy)

The program above makes a copy of listx. The syntax listx[:] is the same as listx[0:len(listx)]. A new list is created in computer memory and the new variable listy is associated to the new list:

listx

[10, 11, 12, 13, 14, 15, 16]

listy

[10, 11, 12, 13, 14, 15, 16]

Here is the output of the print statement:

listy: [10, 11, 12, 13, 14, 15, 16]

The elements of the two lists are exactly the same.

# **1.8 Copying a List**

Can we create a copy of a list by doing this?

listy = listx

The answer is no. The statement above does not create a copy of listx. Instead, it only adds an association of the variable listy to the list already referenced by listx. In other words, both variables listx and listy are associated to the same list in computer memory.

listx

[10, 11, 12, 13, 14, 15, 16]

listy

One way to make a copy of a list is using the copy method. Example:

listx = [10, 11, 12, 13, 14, 15, 16]  
listy = listx.copy()

Another way to make a copy of a list is using the list method. The list method creates a new list. If no argument is passed to it, it creates an empty list. If a list is passed to it as an argument, it will create a copy of that list. Example:

listx = [10, 11, 12, 13, 14, 15, 16]  
listy = list(listx)

listy will be a copy of listx.

# **2. Tuples**

Tuples are similar to lists, except that once a tuple is created, it cannot be altered. The following example shows how to create a tuple and access its elements:

tuple1 = (8, 2, 6, 4)  
print(**"Element 0 of tuple1:"**, tuple1[0])

The following is the output of the program:

Element 0 of tuple1: 8

You cannot change elements of a tuple.

tuple1 = (8, 2, 6, 4)

tuple1[0] = 5

Error message:

TypeError: 'tuple' object does not support item assignment

You cannot add elements to a tuple.

tuple1 = (8, 2, 6, 4)

tuple1.append(5)

Error message:

AttributeError: 'tuple' object has no attribute 'append'

You cannot remove elements from a tuple.

tuple1 = (8, 2, 6, 4)

del(tuple1[0])

Error message:

TypeError: 'tuple' object doesn't support item deletion

You can do concatenation and slicing with tuples because these operations do not alter the original tuples:

tuple1 = (8, 2, 6, 4)  
tuple2 = (7, 3)  
tuple3 = tuple1 + tuple2  
tuple4 = tuple1[0:3]  
print(**"tuple3:"**, tuple3)  
print(**"tuple4:"**, tuple4)

The following is the output of the program:

tuple3: (8, 2, 6, 4, 7, 3)

tuple4: (8, 2, 6)

Tuples are best used for storing data that should remain unchanged throughout the whole program. If necessary, you can convert a tuple to a list and vice versa.

tuple1 = (8, 2, 6, 4)  
print(**"Original tuple:"**, tuple1)  
list1 = list(tuple1)  
list1[0] = 1  
print(**"List after element change:"**, list1)  
tuple1 = tuple(list1)  
print(**"Tuple after element change:"**, tuple1)

The program above first creates a 4-element tuple. It then creates a list and copies the elements from the tuple to it. The syntax list(tuple1) returns a list that has elements exactly the same as tuple1. The program then changes the first element of the list from 8 to 1 and copies the changed list back to a tuple. The syntax tuple(list1) returns a tuple that has elements exactly the same as list1. The following is the output of the program:

Original tuple: (8, 2, 6, 4)

List after element change: [1, 2, 6, 4]

Tuple after element change: (1, 2, 6, 4)

# **3. Sets**

A set is a group of unordered values with no duplicates. It is designed to provide the usual mathematical set operations, including union, intersection, difference and symmetric difference.

To create a set, enclose the elements with a pair of curly braces. Example:

fruit = {'apple', 'banana', 'pear', 'peach'}

A set named fruit is created with four elements.

Sometimes we want to start with an empty set and add elements to it later. To create an empty set, call the set function without passing any argument to it:

fruit = set()

We can use the add method of a set to add an element, and use the remove method to remove an element. Example:

fruit = set() # create empty set  
fruit.add(**'apple'**)  
fruit.add(**'banana'**)  
fruit.add(**'pear'**)  
fruit.add(**'peach'**)  
print(**"Elements in the set:"**, fruit)  
fruit.remove(**'pear'**)  
print(**"Elements in the set after remove:"**, fruit)

Output:

Elements in the set: {'apple', 'peach', 'banana', 'pear'}

Elements in the set after remove: {'apple', 'peach', 'banana'}

We can use the len function to get the length of a set, and use the in operator to test whether a target value is in the set or not. Example:

fruit = {**'apple'**, **'banana'**, **'pear'**, **'peach'**}  
  
length = len(fruit)  
**print**(**'Number of elements in the set:'**, length)  
  
target = input(**"Enter a search target: "**)  
**if** target **in** fruit:  
 **print**(**"Target found in set"**)  
**else**:  
 **print**(**"Target not found in set"**)

Sample test run:

Number of elements in the set: 4

Enter a search target: banana

Target found in set

There are four Python set operations implementing four mathematical set operations: union, intersection, difference and symmetric difference. The union of two sets A and B, denoted as A | B in Python, includes all elements that are either in A or B. The interaction of two sets, denoted as A & B, includes only elements that are in both sets. The difference between A and B, denoted as A - B, includes elements in set A but not in set B. Notice that B - A is not the same as A - B, because B - A gives us the elements in B that are not in A. Finally, the symmetric difference of A and B, denoted as A ^ B, gives us the elements in set A or set B, but not both. Example:

set\_A = {**'apple'**, **'banana'**, **'pear'**, **'peach'**}  
set\_B = {**'orange'**, **'banana'**, **'grape'**}  
  
print(**"A union B:"**, set\_A | set\_B)  
print(**"A intersect B:"**, set\_A & set\_B)  
print(**"A - B:"**, set\_A - set\_B)  
print(**"B - A:"**, set\_B - set\_A)  
print(**"Symmetric difference:"**, set\_A ^ set\_B)

Output of the program:

A union B: {'grape', 'pear', 'peach', 'banana', 'apple', 'orange'}

A intersect B: {'banana'}

A - B: {'pear', 'peach', 'apple'}

B - A: {'grape', 'orange'}

Symmetric difference: {'pear', 'apple', 'orange', 'grape', 'peach'}

If we already have a list or tuple and want to create a set from it, we can pass the list or the tuple to the set function. Example:

fruit\_list = [**'apple'**, **'banana'**, **'pear'**, **'peach'**]  
fruit = set(fruit\_list)  
print(**"Elements in the set:"**, fruit)

Output of the program:

Elements in the set: {'peach', 'pear', 'banana', 'apple'}

If we already have a set and we want to create a list or tuple from it, we can pass the set to the list function or the tuple function. Example:

fruit\_set = {**'apple'**, **'banana'**, **'pear'**, **'peach'**}  
fruit\_list = list(fruit\_set)  
**print**(**"Elements in the list:"**, fruit\_list)  
fruit\_tuple = tuple(fruit\_set)  
**print**(**"Elements in the tuple:"**, fruit\_tuple)

Output of the program:

Elements in the list: ['peach', 'pear', 'apple', 'banana']

Elements in the tuple: ('peach', 'pear', 'apple', 'banana')

Please note that when we create a list from a set, Python places the elements in the list in any order it likes. It is unpredictable and we have no control over it.

# **4. Dictionaries**

In this section, we are learning a new data structure type called dictionary. **Dictionary** is an associative data structure in which items are unordered and accessed by an associated key value. There is no index in dictionary to access data. Instead, we create a key to associate with each value when we add an item to a dictionary. We use the key to access the associated value. For instance, we can use strings such as ‘sun’, ‘mon’, ‘tue’ as keys in a dictionary that stores the daily average temperatures of a week. The following statement creates this dictionary:

daily\_temps = {'sun': 68.8, 'mon': 70.2, 'tue': 67.2, 'wed': 71.8, 'thur': 73.2, 'fri': 75.6, 'sat': 74.0}

Each item in a dictionary includes two parts: a key and a value. The key is separated from its value by a colon (:), the items are separated by commas, and the whole dictionary is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

We use the key to access the value. To display the temperature on Wednesday, we write:

print(daily\_temps['wed'])

The following program creates the temperature dictionary and displays the temperatures on Wednesday and Saturday:

daily\_temps = {**'sun'**: 68.8, **'mon'**: 70.2, **'tue'**: 67.2, **'wed'**: 71.8, **'thur'**: 73.2, **'fri'**: 75.6, **'sat'**: 74.0}  
  
print(**"Temperature on Wednesday:"**, daily\_temps[**'wed'**])  
print(**"Temperature on Saturday:"**, daily\_temps[**'sat'**])

Output of the program:

Temperature on Wednesday: 71.8

Temperature on Saturday: 74.0

Here is another example. Suppose we are storing the names of a team of basketball players in a dictionary. We may use jersey numbers as keys:

players = {17: 'David Fox', 25: 'Jerry Rice', 8: 'George Simmons', 51: 'Larry King', 40: 'Tyler Park'}

The following program creates a dictionary for the basketball players. The program also displays the names of the players and uses the len function to find the number of items in the dictionary:

players = {17: **'David Fox'**, 25: **'Jerry Rice'**, 8: **'George Simmons'**, 51: **'Larry King'**, 40: **'Tyler Park'**}  
  
print(players[17])  
print(players[25])  
print(players[8])  
print(players[51])  
print(players[40])  
print(**"Number of players:"**, len(players))

Output of the program:

David Fox

Jerry Rice

George Simmons

Larry King

Tyler Park

Number of players: 5

The next example creates a dictionary of radio stations:

stations = {89.7:**'WCPE'**, 91.5:**'WUNC'**, 92.3:**'WKRR'**, 92.5:**'WYFL'**}  
  
print(**"89.7 FM:"**, stations[89.7])  
print(**"91.5 FM:"**, stations[91.5])  
print(**"92.3 FM:"**, stations[92.3])  
print(**"92.5 FM:"**, stations[92.5])  
print(**"Number of stations:"**, len(stations))

Output of the program:

89.7 FM: WCPE

91.5 FM: WUNC

92.3 FM: WKRR

92.5 FM: WYFL

Number of stations: 4

## 4.1 Creating a Dictionary with the dict Function

In addition to using the syntax we saw earlier, we can also use the dict function to create a dictionary. If no argument is passed to the dict function, an empty dictionary is created. Example:

stations = dict()  
print(**"Number of stations:"**, len(stations))

Output of the program:

Number of stations: 0

A dictionary is created if a list of the following key-value pairs is passed to the dict function:

[[key1, value1], [key2, value2], [key3, value3], …]

Example:

station\_list = [[89.7,**'WCPE'**], [91.5,**'WUNC'**], [92.3,**'WKRR'**], [92.5,**'WYFL'**]]  
stations = dict(station\_list)  
  
print(**"89.7 FM:"**, stations[89.7])  
print(**"91.5 FM:"**, stations[91.5])  
print(**"92.3 FM:"**, stations[92.3])  
print(**"92.5 FM:"**, stations[92.5])  
print(**"Number of stations:"**, len(stations))

Output of the program:

89.7 FM: WCPE

91.5 FM: WUNC

92.3 FM: WKRR

92.5 FM: WYFL

Number of stations: 4

## 4.2 Adding or Modifying an Item

You can update a dictionary by adding a new entry or modifying an existing entry as shown below in the example:

stations = {89.7:**'WCPE'**, 91.5:**'UNCW'**, 92.3:**'WKRR'**, 92.5:**'WYFL'**}  
  
stations[91.5] = **'WUNC'** *# update existing item*stations[88.9] = **'WSHA'** *# add new item*print(**"88.9 FM:"**, stations[88.9])  
print(**"89.7 FM:"**, stations[89.7])  
print(**"91.5 FM:"**, stations[91.5])  
print(**"92.3 FM:"**, stations[92.3])  
print(**"92.5 FM:"**, stations[92.5])  
print(**"Number of stations:"**, len(stations))

Output of the program:

88.9 FM: WSHA

89.7 FM: WCPE

91.5 FM: WUNC

92.3 FM: WKRR

92.5 FM: WYFL

Number of stations: 5

## 4.3 Deleting an Item

You can use keys to remove individual dictionary items or clear the entire contents of a dictionary. Example:

stations = {89.7:**'WCPE'**, 91.5:**'UNCW'**, 92.3:**'WKRR'**, 92.5:**'WYFL'**}  
  
**del** stations[91.5] *# delete dictionary item*print(**"Number of stations after del:"**, len(stations))  
  
stations.clear() *# delete all items*print(**"Number of stations after clear:"**, len(stations))

Output of the program:

Number of stations after del: 3

Number of stations after clear: 0

You can also use values to remove dictionary items. Example:

stations = {89.7:'WCPE', 91.5:'WUNC', 92.3:'WKRR', 92.5:'WYFL'}

target = input("Enter a station to remove: ")

key\_to\_remove = ""

for key in stations:

if stations[key] == target:

key\_to\_remove = key

if key\_to\_remove != "":

del stations[key\_to\_remove]

print('Station removed')

print('Updated set:', stations)

else:

print('Station not found')

Sample test run:

Enter a station to remove: WUNC

Station removed

Updated set: {89.7: 'WCPE', 92.3: 'WKRR', 92.5: 'WYFL'}

## 4.4 Testing Membership

You can use the in operator to test whether a key exists in a dictionary. Example:

stations = {89.7:**'WCPE'**, 91.5:**'UNCW'**, 92.3:**'WKRR'**, 92.5:**'WYFL'**}  
  
target = float(input(**"Enter a station number: "**))  
  
**if** target **in** stations:  
 print(**"Station found in dictionary"**)  
**else**:  
 print(**"Station not found in dictionary"**)

Sample test run:

Enter a station number: 92.5

Station found in dictionary

You can also test whether a value exists in a dictionary. Example:

stations = {89.7:'WCPE', 91.5:'WUNC', 92.3:'WKRR', 92.5:'WYFL'}

target = input("Enter a station name: ")

in\_dictionary = False

for key in stations:

if stations[key] == target:

in\_dictionary = True

if in\_dictionary == True:

print("The station is in the dictionary")

else:

print("The station is not in the dictionary")

Sample test run:

Enter a station name: WUNC

The station is in the dictionary

## 4.5 Iterating over a Dictionary

If we use a for loop to iterate over a dictionary, we get the keys, not the values. Example:

players = {17: 'David Fox', 25: 'Jerry Rice', 8: 'George Simmons', 51: 'Larry King', 40: 'Tyler Park'}

for element in players:

print(element)

Output of the program:

17

25

8

51

40

We can modify the program above to get the values. Example:

players = {17: 'David Fox', 25: 'Jerry Rice', 8: 'George Simmons', 51: 'Larry King', 40: 'Tyler Park'}

for element in players:

print(players[element])

Output of the program:

David Fox

Jerry Rice

George Simmons

Larry King

Tyler Park

## 4.6 Creating Lists from a Dictionary

Sometimes we want to extract the keys or values from a dictionary and store them in lists. The dictionary method keys returns a sequence of the dictionary’s keys. Similarly, the values method returns a sequence of the dictionary’s values, while the items method returns a sequence of (key, value) tuples. These sequences can then be converted to lists using the list function. The following is an example:

stations = {89.7:**'WCPE'**, 91.5:**'UNCW'**, 92.3:**'WKRR'**, 92.5:**'WYFL'**}  
  
stations\_nums = list(stations.keys())  
print(**"Station Keys:"**)  
**for** key **in** stations\_nums:  
 print(key)

print()  
stations\_names = list(stations.values())  
print(**"Station Names:"**)  
**for** name **in** stations\_names:  
 print(name)  
  
print()

stations\_tuples = list(stations.items())  
print(**"Key-Name Tuples:"**)  
**for** tup **in** stations\_tuples:  
 print(tup)

Output of the program:

Station Keys:

92.5

89.7

91.5

92.3

Station Names:

WYFL

WCPE

UNCW

WKRR

Key-Name Tuples:

(92.5, 'WYFL')

(89.7, 'WCPE')

(91.5, 'UNCW')

(92.3, 'WKRR')