# Chapter 4: Data Structures

## Contents



## Syllabus Learning Outcomes

2.2 Programming Elements and Constructs

Use programming language elements and constructs to write recursive and non-recursive programs to solve a variety of problems.

2.2.1 Understand the different types: integer, real, char, string and Boolean and initialise arrays (1-dimensional and 2-dimensional).

# 1 Lists

Lists are the most commonly used data structure in Python. Lists are used to store multiple items in a single variable. List items are ordered, mutable and allow duplicate values.

A list collection is mutable means its items can be added, updated and removed. Each of these data can be accessed by calling its index value.

## 1.1 Create

Lists are created using square brackets. There are a few ways to initialise a list.

```
# initialise a list

# creating an empty list using square brackets
nums = []
print(nums)

# creating an empty list using list constructor
nums = list()
print(nums)

# create a list with elements
nums = [1, 2, 3, 4, 5]
print(nums)
```

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

List items can be of any data type. A list can also contain different data types.

```
# list can be of any data types
nums = list(range(10))
print(nums)

characters = list("spiderman")
print(characters)

fruits = ["apple", "banana", "orange"]
print(fruits)

# list can also be of mixed data types
list1 = ["abc", 34, True, 40, "male"]
print(list1)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
['s', 'p', 'i', 'd', 'e', 'r', 'm', 'a', 'n']
['apple', 'banana', 'orange']
['abc', 34, True, 40, 'male']
```

A list can also contain sublists, which is known as a nested list.

```
# Example of nested list

list2 = [[1,"apple"],[2,"orange"],[3,"pear"]]

print(list2[0])
print(list2[0][0])
print(list2[0][1])

```

```
[1, 'apple']
1
apple
```

## 1.2 Add list items

### 1.2.1 Append

The `append( )` method is used to append an element at the end of the list.

```
# append adds an item to the end of the list
num = [1, 2, 3, 4, 5]
num.append(6)
print(num)


num = [1, 2, 3, 4, 5]
num.append([6,7,8])
print(num)
```

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

### 1.2.2 Insert

The `insert()` method inserts an item at the specified index.

```
# insert
fruits = ["apple", "banana", "orange"]
fruits.insert(2, "cherry")
print(fruits)
```

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

### 1.2.3 Extend

A list can also be extended with items from another list using `extend()` method. It will modify the first list. The resultant list will contain all the elements of the lists that were added, i.e. the resultant list is NOT a nested list.

```
# extending the list
num = [1, 2, 3, 4, 5]
num.extend([6, 7, 8])
print(num)
```

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

### 1.2.4 Using operator

Two lists can also be join together simply using + operator.

```
# adding 2 lists together
num = [1,2,3,4,5]
num = num + [6,7,8]
print(num)
```

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

Similar to String, we can repeat a list multiple times with * operator.

```
num = [1, 2, 3]
num = num*5
print(num)
```

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

## 1.3 Access list items

List items are indexed and you can access them by referring to the index number. The first item has index 0.

```
fruits = ['apple', 'banana', 'cherry', 'durian']
print(fruits[0]) #print element at index 0
print(fruits[1]) #print element at index 1
```

```
apple
banana
```

We can also access the elements in the list using negative indexing. That is the last element has an index of -1, and second last element has index of -2 and so on.

```
fruits = ['apple', 'banana', 'cherry', 'durian']
print(fruits[-1]) #print last element
print(fruits[-2]) #print second last element
```

```
durian
cherry
```

To iterate through a list, we can use for loop.

```
fruits = ['apple', 'banana', 'cherry', 'durian']

for item in fruits:
    print(item)
```

```
apple
banana
cherry
durian
```

Similar to Strings, we can use the len() method to get the number of elements in a given list.

```
fruits = ['apple', 'banana', 'cherry', 'durian']
print(len(fruits))
```

```
4
```

Indexing was only limited to accessing a single element. Slicing on the other hand is accessing a sequence of data inside the list.

Slicing is done by defining the index values of the first element and the last element from the parent list that is required in the sliced list, similar to Strings.

sub = num[a : b]
sub = num[a : ]
sub = num[: b]
sub = num[:]

if both a and b are specified, a is the first index, b - 1 is the last index. Note that the upper bound index is NOT inclusive.

if b is omitted, it will slice till last element.
if a is omitted, it will starts from first element.
if neither a or b is specified, it is effectively copy the whole list

```
# Example of slicing
fruits = ['apple', 'banana', 'cherry', 'durian']
print(fruits[1:3]) # prints elements from index 1 to 2
print(fruits[:3]) # prints elements from index 0 to 2
print(fruits[1:len(fruits)]) # print elements from index 1 to last element
print(fruits[1:]) # print elements from index 1 to last element
print(fruits[1:-1]) # print elements from index 1 to second last element
```

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

The `index()` method returns the position at the first occurrence of the specified value. If there are multiple items of the same value, only the first index value of that item is returned. You can add 2nd argument to start searching from a particular index onwards.

```
# example of using index() method
fruits = ['apple', 'banana', 'cherry', 'durian', 'banana']
print(fruits.index("cherry"))
print(fruits.index("banana"))
print(fruits.index("banana",2))

```

```
2
1
4
```

### 1.4 Update

An item in the list can be updated by its index value.

```
fruits = ['apple', 'banana', 'cherry', 'durian']
fruits[1] = 'blackcurrent'
print(fruits)
```

```
['apple', 'blackcurrent', 'cherry', 'durian']
```

You can update more than one item by specifying a range of index numbers where you want to insert the new items.

```
fruits = ['apple', 'banana', 'cherry', 'durian']
fruits[1:3] = ['blackcurrent', 'watermelon']
print(fruits)
```

```
['apple', 'blackcurrent', 'watermelon', 'durian']
```

If you insert more items than you replace/update, the new items will be inserted where you specified, and the remaining items will move accordingly.

```
fruits = ['apple', 'banana', 'cherry', 'durian']
fruits[1:3] = ['blackcurrent', 'watermelon', 'strawberry']
print(fruits)
```

```
['apple', 'blackcurrent', 'watermelon', 'strawberry', 'durian']
```

## 1.5 Delete

### 1.5.1 Remove item by value

The remove( ) method remove an item based on its value. If there are multiple items of same value, it will only remove 1st item.

It will throw an exception if the value is not found in the list.

```
num = [10,20,30]
num.remove(20)
print(num)
num.remove(40)
```

```
[10, 30]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-33-caf5ef45a1ad> in <module>()
      2 num.remove(20)
      3 print(num)
----> 4 num.remove(40)

ValueError: list.remove(x): x not in list
```

### 1.5.2 Remove item by index

The pop( ) method removes the element at the specified position. If no position is specified, last item is removed by default. The pop() method returns removed value.

```
num = [1, 2, 3, 4, 5]
num.pop() #last item is removed
print(num)

num = [1, 2, 3, 4, 5]
x = num.pop(1) # remove and return item at index 1
print(num)
print(x)
```

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

### 1.5.3 Del keyword

The **del** keyword is used to delete objects. In Python, everything is an object, so the del keyword can also be used to delete variables, lists, or parts of a list etc.

```
num = [1, 2, 3, 4, 5]

del num[2] #del element at index 2
print(num)
del num
print(num)
```

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

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-36-fc63cb7c0723> in <module>()
      4 print(num)
      5 del num
----> 6 print(num)

NameError: name 'num' is not defined
```

**del** num deleted the num list entirely, hence an error is encountered when the `print(num)` command is executed.

### 1.5.4 Clear a list

To clear all elements in a list, use its clear() method.

```
num = [1,2,3]
print(num)
num.clear()
print(num)
```

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

## 1.6 Working with List

### 1.6.1 Min, Max and Sum

If the list consists of all integer elements, the min( ), max( ) and sum() gives the minimum item, maximum item and total sum value of the list.

```
num = [0,1,2,3,4,5,6,7,8,9]
print(min(num))
print(max(num))
print(sum(num))
```

```
0
9
45
```

For a list with elements as string, the max( ) and min( ) is still applicable. The max( ) method would return a string element whose ASCII value is the highest. The min( ) method is used to return the lowest. Take note that only the first index of each element is considered each time and if they value is the same then second index considered so on and so forth.

```
poly = ['np','sp','tp','rp','nyp']
print(min(poly))
print(max(poly))
```

```
np
tp
```

### 1.6.2 Reversing

The entire elements present in the list can be reversed by using the `reverse()` method.

```
num = [1, 2, 3, 4, 5, 6, 7, 8, 9]
num.reverse()
print(num)
```

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

### 1.6.3 Sorting

List has a built in method sort( ) to arrange the elements in ascending order by default. For descending order, specify the named argument reverse = True.

```
num = [24, 9, 65, 43, 13]
num.sort() # numbers are sorted in ascending order by default
print(num)
num.sort(reverse = True) # adding reverse = True will sort list in descending order
print(num)
```

```
[9, 13, 24, 43, 65]
[65, 43, 24, 13, 9]
```

Note that both the `sort()` and `reverse()` methods mentioned above will alter the original sequence of items in the list.

The `sorted()` method sorts the given sequence either in ascending order or in descending order and returns the sorted list. This method does not effect the original sequence of items in the list.

```
num = [24, 9, 65, 43, 13]
x = sorted(num)
y = sorted(num, reverse = True)
print(x)
print(y)
print(num) #the original list is not sorted
```

```
[9, 13, 24, 43, 65]
[65, 43, 24, 13, 9]
[24, 9, 65, 43, 13]
```

### 1.6.4 Membership and searching

You can use the in (not in) statement for membership of an item in a list.

```
names = ['duck', 'chicken', 'goose']
print('duck' in names)
print('turkey' not in names)
```

```
True
True
```

The count() method can count the occurrence of a particular item in a list.

```
num = [1, 2, 3, 4, 5, 5, 5, 8, 9]
print(num.count(5))
```

```
3
```

## 1.7 Copying 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.

```
list1= [0,1,2,3]
list2 = list1
print(list1 == list2)
print(list1 is list2)

# Modify a list, both lists are affected
list1.pop()
print(list1)
print(list2)
```

```
True
True
[0, 1, 2]
[0, 1, 2]
```

### 1.7.1 By Slicing

There are ways to make a copy, one way is to copy by slicing.
Slicing without start and end index returns all items in the list.

```
1  list1= [0, 1, 2, 3]
2  list2 = list1[:]
3  print(list1 == list2)
4  print(list1 is list2)
5
6  # Modify a list, the other is not affected
7  list1.pop()
8  print(list1)
9  print(list2)
```

```
True
False
[0, 1, 2]
[0, 1, 2, 3]
```

### 1.7.2 By Constructor

list() is a constructor function to create a list. When it accepts another list as argument, it creates a duplicate list.

```
1  list1= [0, 1, 2, 3]
2  list2 = list(list1)
3  print(list1 == list2)
4  print(list1 is list2)
5
6  # Modify a list, the other is not affected
7  list1.pop()
8  print(list1)
9  print(list2)
```

```
True
False
[0, 1, 2]
[0, 1, 2, 3]
```

### 1.7.3 The copy() method

The copy() method copies a list into another list.

```
list1= [0, 1, 2, 3]
list2 = list1.copy()
print(list1 == list2)
print(list1 is list2)

# Modify a list, the other is not affected
list1[0] = 10
list2[0] = 20
print(list1)
print(list2)
```

```
True
False
[10, 1, 2, 3]
[20, 1, 2, 3]
```

list2 is stored at different memory location, which is confirmed by values of id() functions. The id() function returns the address of the list. Modifying list2 doesn't affect the values in list1.

```
print(id(list1), id(list2))
```

```
2369564014280 2369564063688
```

Do note that the copy() method only performs shallow copy. It copies the list itself, which contains references to the objects in the list. If an object itself is mutable, change in this object will be reflected in both lists. An example would be nested list.

```
# copying a nested list using copy()
list1 = [1,[2,3,4],5]
list2 = list1.copy()
print("List1: ",list1)
print("List2: ",list2)
print()

list2[1][1] = 9
print("List1: ",list1)
print("List2: ",list2)
print()

list2.append(6)
print("List1: ",list1)
print("List2: ",list2)
```

```
List1:  [1, [2, 3, 4], 5]
List2:  [1, [2, 3, 4], 5]

List1:  [1, [2, 9, 4], 5]
List2:  [1, [2, 9, 4], 5]

List1:  [1, [2, 9, 4], 5]
List2:  [1, [2, 9, 4], 5, 6]
```

For situations like the above, the copy module provides a function deepcopy() to perform deep copy of objects, such that it will also create a duplicate for the nested objects.

```
from copy import deepcopy

list1 = [1,[2,3,4],5]
list2 = deepcopy(list1)
print("List1: ",list1)
print("List2: ",list2)
print()

list2[1][1] = 9 # modifying list2
print("List1: ",list1)
print("List2: ",list2)
print()

list2.append(6) # modifying list2
print("List1: ",list1)
print("List2: ",list2)
print()
```

```
List1:  [1, [2, 3, 4], 5]
List2:  [1, [2, 3, 4], 5]

List1:  [1, [2, 3, 4], 5]
List2:  [1, [2, 9, 4], 5]

List1:  [1, [2, 3, 4], 5]
List2:  [1, [2, 9, 4], 5, 6]
```

## 1.8 List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list. The syntax is as follows:

```
newlist = [expression for item in iterable if condition == True]
```

We can use either a for or while loop to create a list of numbers from 0 to 9.

```
1  # Creating a list of numbers from 0 to 9
2
3  nums = []
4  for num in range(10):
5      nums.append(num)
6
7  print(nums)
```

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

With list comprehension, we can create the same list of numbers 0 to 9 in one simple statement.

```
# Example of using basic list comprehension
# to create a list of numbers from 0 to 9

nums = [x for x in range(10)]
print(nums)
```

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

We can create a list of even numbers by adding in a condition. The condition is like a filter that only accepts the items that valuate to True.

```
# Adding a condition
# to create a list of even numbers from 0 to 9

nums = [x for x in range(10) if x%2 == 0]
print(nums)
```

```
[0, 2, 4, 6, 8]
```

The expression is the outcome, which you can manipulate it before you append it into the list.

```
# double the even values before adding into the list

nums = [x*2 for x in range(10) if x%2 == 0]
print(nums)
```

```
[0, 4, 8, 12, 16]
```

The expression can also contain conditions, not like a filter, but as a way to manipulate the outcome.

```
# create a list of numbers such that
# the number is doubled when the number is even
# the number is tripled when the number is odd
# where the number is from 0 to 9

nums = [x*2 if x%2 == 0 else x*3 for x in range(10) ]
print(nums)
```

```
[0, 3, 4, 9, 8, 15, 12, 21, 16, 27]
```

# 2 Tuples

Tuple behaves like a list but it is **immutable**. Tuples are written with round brackets.

```
# initialise a tuple

# creating an empty tuple using round brackets
nums = ()
print(nums)

# creating an empty list using tuple constructor
nums = tuple()
print(nums)

# create a tuple with elements
nums = (1, 2, 3, 4, 5)
fruits = ("apples", "bananas", "oranges")
print(nums)
print(type(nums))
print(fruits)
```

```
()
()
(1, 2, 3, 4, 5)
<class 'tuple'>
('apples', 'bananas', 'oranges')
```

To create a tuple containing a single item, you have to include a comma; else, Python will not recognise it as a tuple.

```
# creating a tuple with only 1 item

nums = (1)
print(type(nums))

nums = (1,)
print(type(nums))
```

```
<class 'int'>
<class 'tuple'>
```

Tuples are immutable, meaning that we cannot change, add or remove items after the tuple has been created.

```
nums = (1, 2, 3, 4, 5)
nums[0] = 100
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-992d5c4524b8> in <module>()
      1 nums = (1, 2, 3, 4, 5)
----> 2 nums[0] = 100

TypeError: 'tuple' object does not support item assignment
```

```
1  nums = (1, 2, 3, 4, 5)
2
3  del nums[0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-aa31e1f960cf> in <module>()
      1 nums = (1, 2, 3, 4, 5)
      2
----> 3 del nums[0]

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

We can create a list from a tuple using type casting and vice versa.

```
1  nums1 = (1, 2, 3, 4, 5)
2  print(type(nums))
3
4  nums2 = list(nums1)
5  print(nums2)
6  print(type(nums2))
7
8  nums3 = tuple(nums2)
9  print(type(nums3))
```

```
<class 'tuple'>
[1, 2, 3, 4, 5]
<class 'list'>
<class 'tuple'>
```

Tuples respond to all of the general sequence operations we used on strings or lists in the prior chapter.

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

## 2.1 Unpacking Tuples

When we create a tuple, we normally assign values to it. This is called "packing" a tuple. In Python, we are allowed to extract the values back into variables. This is called "unpacking" a tuple.

```python
# unpacking one-dimensional tuple
nums = (1, 2, 3)
a,b,c = nums
print(a)
print(b)
print(c)
```

```
1
2
3
```

```python
# unpacking nested tuples
nums = (1, 2, (3,4))
a,b,c = nums
print(a)
print(b)
print(c)
print()

nums = (1, 2, (3,4))
a,b,(c,d) = nums
print(a)
print(b)
print(c)
print(d)
print()
```

```
1
2
(3, 4)

1
2
3
4
```

# 3 Dictionaries

## 3.1 Create

Dictionaries are used to store items in key:value pairs. Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces.

```
# empty dictionary
dict1 = {}
print(dict1)

dict2 = dict()
print(dict2)

# dictionary with mixed data type
dict3 = {'name': 'John', 'age': 7, 'family': ['Father', 'Mother']}
print(dict3)
```

```
{}
{}
{'name': 'John', 'age': 7, 'family': ['Father', 'Mother']}
```

Dictionary items are unordered (in Python 3.6 and earlier), changeable, and does not allow duplicates. 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. Duplicate values will overwrite existing values

```
# dictionary does not allow duplicates
dict3 = {'name': 'John', 'age': 7, 'family': ['Father', 'Mother'], 'family':['Brother']}
print(dict3)
```

```
{'name': 'John', 'age': 7, 'family': ['Brother']}
```

## 3.2 Access

Dictionary items can be accessed by using the corresponding keys.

```
# accessing values using corresponding key
dict3 = {'name': 'John', 'age': 7, 'family': ['Father', 'Mother']}
print("Name: ", dict3['name'])
print("Age: ", dict3['age'])
```

```
Name:  John
Age:  7
```

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error.

```
# accessing values using corresponding key
dict3 = {'name': 'John', 'age': 7, 'family': ['Father', 'Mother']}
print("Name: ", dict3['name'])
print("Age: ", dict3['age'])
print("Address", dict3['address'])
```

```
Name:  John
Age:  7

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-6-238ae73aac70> in <module>()
      3 print("Name: ", dict3['name'])
      4 print("Age: ", dict3['age'])
----> 5 print("Address", dict3['address'])

KeyError: 'address'
```

We can use the method get() to access the values with the corresponding keys. This method is more forgiving as it does not throw an error if key does not exist. However, it returns a None object.

```
# accessing values using get()
dict3 = {'name': 'John', 'age': 7, 'family': ['Father', 'Mother']}
print("Name: ", dict3.get('name'))
print("Age: ", dict3.get('age'))
print("Address", dict3.get('address'))
```

```
Name:  John
Age:  7
Address None
```

The keys() method will return a list of all the keys in the dictionary. The values() method will return a list of all the values in the dictionary. The items() method will return each item in a dictionary, as tuples in a list.

```
# keys() and values()
dict3 = {'name': 'John', 'age': 7, 'family': ['Father', 'Mother']}
print(dict3.keys())
print(dict3.values())
print(dict3.items())
```

```
dict_keys(['name', 'age', 'family'])
dict_values(['John', 7, ['Father', 'Mother']])
dict_items([('name', 'John'), ('age', 7), ('family', ['Father', 'Mother'])])
```

To determine if a specified key is present in a dictionary, we can use the in keyword.

```
dict3 = {'name': 'John', 'age': 7, 'family': ['Father', 'Mother']}
print('name' in dict3)
print('address' in dict3)
```

```
True
False
```

We can also use in to access all keys in the dictionary and the corresponding values.

```
dict3 = {'name': 'John', 'age': 7, 'family': ['Father', 'Mother']}

for k in dict3:
    print(k, dict3[k])
```

```
name John
age 7
family ['Father', 'Mother']
```

We can also print all key:value pair by using items().

```
dict3 = {'name': 'John', 'age': 7, 'family': ['Father', 'Mother']}

for k,v in dict3.items():
    print(k, v)
```

```
name John
age 7
family ['Father', 'Mother']
```

### 3.3 Update

We can change the value of a specific item by referring to its key.

```
# update value by using its key
food = {'a':'Apple', 'b': 'Berries',
        'c': 'Cereal', 'd': 'Doughnut'}

food['b'] = 'Banana'
print(food)
```

```
{'a': 'Apple', 'b': 'Banana', 'c': 'Cereal', 'd': 'Doughnut'}
```

We can also change the values of a specific item by using the update() method.

```
# update using update() method
food = {'a':'Apple', 'b': 'Berries',
        'c': 'Cereal', 'd': 'Doughnut'}

food.update({'c': 'Carrot'})
print(food)
```

```
{'a': 'Apple', 'b': 'Berries', 'c': 'Carrot', 'd': 'Doughnut'}
```

### 3.4 Add

Adding an item to the dictionary can be done by using a new index key and assigning a value to it.

```
# add a new food
food = {'a':'Apple', 'b': 'Berries',
        'c': 'Cereal', 'd': 'Doughnut'}

food['e'] ='Eggplant'
print(food)
```

```
{'a': 'Apple', 'b': 'Berries', 'c': 'Cereal', 'd': 'Doughnut', 'e': 'Eggplant'}
```

The update() method can also be used to add new items into dictionaries.

```
# add new items using update()
food = {'a':'Apple', 'b': 'Berries',
        'c': 'Cereal', 'd': 'Doughnut'}

food.update({'e':'Eggplant', 'f':'Fish'})
print(food)
```

```
{'a': 'Apple', 'b': 'Berries', 'c': 'Cereal', 'd': 'Doughnut', 'e': 'Eggplant', 'f': 'Fish'}
```

### 3.5 Delete

An item can be deleted by using the pop() method. The method removes an item by its key and returns the value. It throws an exception if key is not found.

```
# delete item by using pop()
food = {'a':'Apple', 'b': 'Berries',
        'c': 'Cereal', 'd': 'Doughnut'}
food.pop('c')
print(food)
food.pop('e')
```

```
{'a': 'Apple', 'b': 'Berries', 'd': 'Doughnut'}

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-29-0bea29e7add1> in <module>()
      4 food.pop('c')
      5 print(food)
----> 6 food.pop('e')

KeyError: 'e'
```

The del keyword removes the item with the specified key name.

```
# delete item by using del
food = {'a':'Apple', 'b': 'Berries',
        'c': 'Cereal', 'd': 'Doughnut'}
del food['d']
print(food)
```

```
{'a': 'Apple', 'b': 'Berries', 'c': 'Cereal'}
```

The del keyword can also delete the dictionary completely.

```
# delete dictionary by using del
food = {'a':'Apple', 'b': 'Berries',
        'c': 'Cereal', 'd': 'Doughnut'}
del food
print(food)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-31-2fc0c25beec7> in <module>()
      3         'c': 'Cereal', 'd': 'Doughnut'}
      4 del food
----> 5 print(food)

NameError: name 'food' is not defined
```

The clear() method empties the dictionary.

```
# empty dictionary using clear()
food = {'a':'Apple', 'b': 'Berries',
        'c': 'Cereal', 'd': 'Doughnut'}
food.clear()
print(food)
```

```
{}
```