Chapter 6: Lists, Tuples and Dictionaries Oh My!

Contents

Time required: 90 minutes

DRY

Don't **R**epeat **Y**ourself

Visualize and Debug Programs

The website www.pythontutor.com helps you create visualizations for the code in all the listings and step through those programs one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. This will help you to better understand the behavior of the programs we are working on.

This is a great way to debug your code. You can see the variables change as you step through the program.

www.pythontutor.com

Lists and Tuples

Lists and tuples are ordered collections or sequences of data items. Many operations will work with either. The major difference is that lists are mutable, tuples are immutable. This means that once created, a list can be changed, a tuple cannot.

A List is a Sequence

We need to get thirty test scores from a user and do something with them. We want to put them in order. We could create thirty variables, score1, score2, . . . , score30, but that would be very tedious. To put the scores in order would be extremely difficult. The solution is to use lists.

A list is a sequence of any type of values. The values in a list are called elements or sometimes items.

The following is a list named **a_list** which contains string values.

```
a_list = ["p", "y", "t", "h", "o" "n"]
```

This diagram shows how this list is organized and accessed. List index positions start at 0.

	← length = 6 →					
	"p"	"y"	"t"	"h"	"o"	"n"
Index	0	1	2	3	4	5
Negative index	-6	-5	-4	-3	-2	-1

The program below shows how to create a list and access items in a list.

```
# simple_list.py
# Create a simple list with three integers
L = [1, 2, 3]
# Print the list
print(L)
# Access the first element of the list
print(L[0])
# Access the second element of the list
print(L[1])
# Access the last element of the list
print(L[-1])
```

Example run for both programs:

```
Please enter 5 numbers:
Enter number 1: 1
Enter number 2: 2
Enter number 3: 3
Enter number 4: 4
Enter number 5: 5
Average: 3.0
```

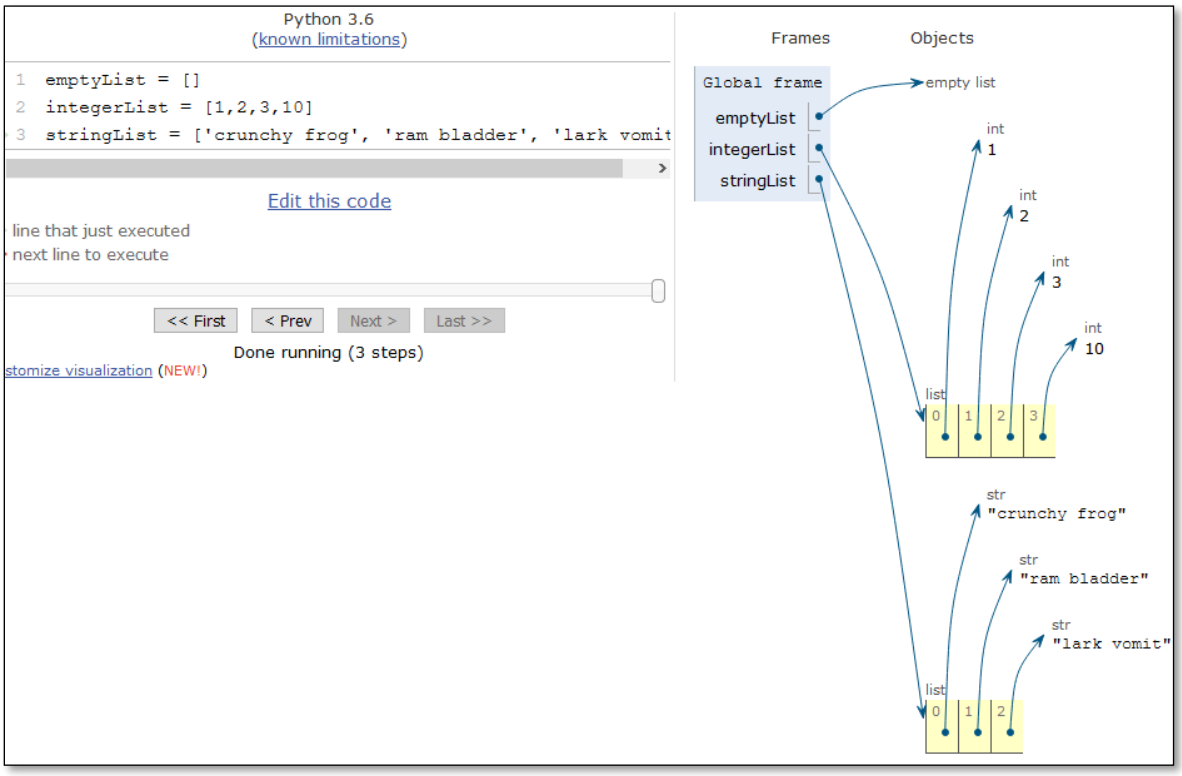
The list version of the average program can be expanded as much as we want. Change the `NUMBER_OF_ENTRIES` constant to whatever you wish, the list will take care of it without any extra lines of code.

Creating Lists

There are several ways to create a new list; the simplest is to enclose the elements in square brackets. An empty list is `[]`. It is the list equivalent of 0 or `''`.

```
empty_list = []
```

```
integer_list = [1, 2, 3, 10]
string_list = ["crunchy frog", "ram bladder", "lark vomit"]
```



The first example is an empty list. The second is a list of four integers. The third is a list of three strings.

The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

```
monty_list = ["spam", 2.0, 5, [10, 20]]
```

Long Lists: If you have a long list to enter, you can split it across several lines, like below:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39, 40]
```

Input: We can use eval(input()) to allow the user to enter a list. **eval** let's Python decide what type of values are being input. Here is an example:

```
user_list = eval(input("Enter a list: "))
print(f"The first element is {user_list[0]} ")
```

Example program run:

```
Enter a list: [1,2,3]
The first element is  1
```

Printing Lists: You can use the print function to print the entire contents of a list.

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

[1, 2, 3]
```

Accessing a List

You can access individual items in a list by the index number. Remember, the index of a list starts at 0.

```
# Create a list of strings
cheeses = ["1. Cheddar", "2. Swiss", "3. Feta"]
# Print the list
for cheese in cheeses:
    print(cheese)

# Prompt the user for a choice
choice = int(input("Enter the number of the cheese you wish: "))

# Print the user choice
print(cheeses[choice - 1])
```

Example run:

```
1. Cheddar
2. Swiss
3. Feta
Enter the number of the cheese you wish: 3
3. Feta
```

Lists are Mutable

The syntax for accessing the elements of a list is the bracket operator. The expression inside the brackets specifies the index. Indices start at 0:

```
cheeses = ["Cheddar", "Swiss", "Feta"]
print(cheeses[0])

Cheddar
```

Lists are mutable. You can change the order of items in a list or reassign an item in a list.

When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
numbers = [17, 123]
numbers[1] = 5
print(numbers)

[17, 5]
```

The number 1 element of the numbers list, which used to be 123, is now 5.

Values from a list can be used just like a variable. You can use F-strings to create a message.

```
# print_message.py
dogs = ["Affenpinscher", "Afghan Hound", "Akita"]
message = f"My dog is an {dogs[0]}"
print(message)

My dog is an Affenpinscher
```

You can think of a list as a relationship between indices and elements. This relationship is called a mapping; each index "maps to" one of the elements.

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an IndexError.
- If an index has a negative value, it counts backward from the end of the list.

Changing Lists

To change the value in location 2 of L to 100, we simply say L[2]=100. If we want to insert the value 100 into location 2 without overwriting what is currently there, we can use the insert method. To delete an entry from a list, we can use the del operator. Some examples are shown below. Assume L=[6, 7, 8] for each operation.

Operation	New L	Description
L[1]=9	[6, 9, 8]	replace item at index 1 with 9
L.insert(1,9)	[6, 9, 7, 8]	insert a 9 at index 1 without replacing
del L[1]	[6, 8]	delete second item
del L[:2]	[8]	delete first two items

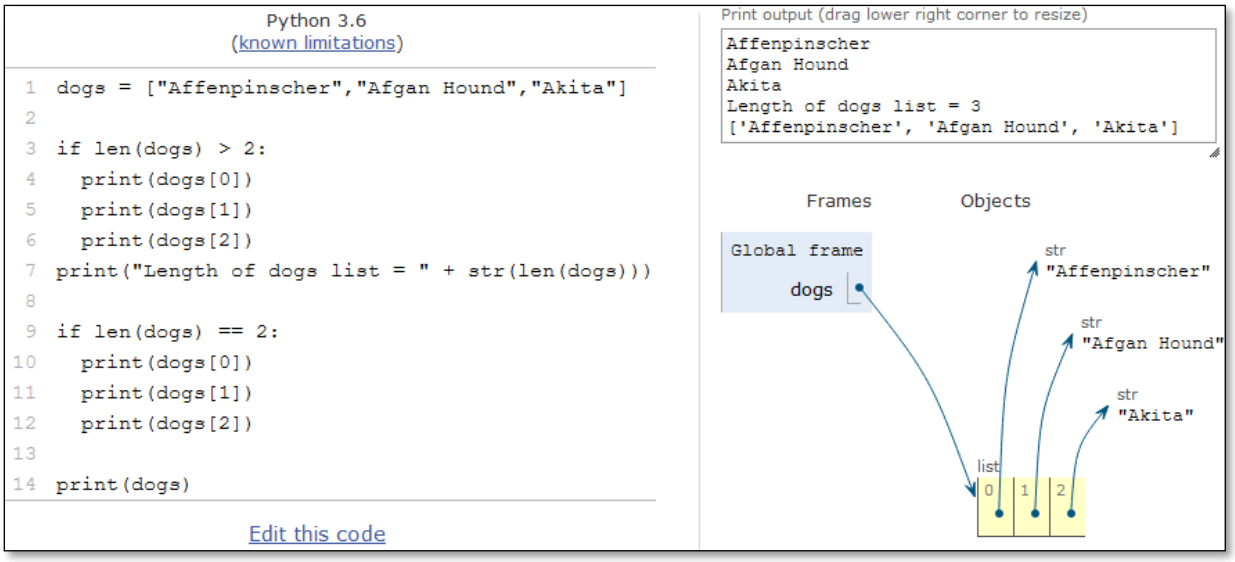
Example of if Statement with Lists

```
dogs = ["Affenpinscher", "Afghan Hound", "Akita"]

if len(dogs) > 2:
  print(dogs[0])
  print(dogs[1])
  print(dogs[2])
print(f"Length of dogs list {len(dogs)} ")

if len(dogs) == 2:
  print(dogs[0])
  print(dogs[1])
  print(dogs[2])

print(dogs)
```



1. The listing begins by creating and populating a list containing the names of three breeds of dogs. Then it tests to see if the length of the list is greater than 2, which it is.

2. Because the test returns True, each element in the list is printed in sequence by the three indented print statements.

3. After that, the length of the list is printed by the non-indented print statement. This requires that the numeric length of the list be obtained and converted to a string for printing.
4. The built-in function named **len** is called here (and in the conditional clause of the if statement) to get the length of the list.
5. The built-in function named str is called to convert the numeric length of the list into a string for concatenation with another string for printing.
6. The process is repeated except instead of testing the length of the list for greater than 2, it is tested for equal to 2. This test returns False, which causes the indented print statements in the body of the if statement to be skipped.
7. After that, the list is printed producing the last line of text shown.

Making Copies of Lists

Making copies of lists is a little tricky due to the way Python handles lists. Say we have a list L and we want to make a copy of the list and call it M. The expression M=L will not work. Do the following in place of M=L:

```
M = L[:]
```

Traversing a List with a Loop

The most common way to traverse the elements of a list is with a for loop. The syntax is the same as for strings:

This loop is useful when you need to print out all values in a list.

```
for cheese in cheeses:
    print(cheese)
cheddar
colby
swiss
```

If we want the index number and the element, we use the enumerate method.

```
for index, cheese in enumerate(cheeses):
    print(index, cheese)
0 cheddar
1 colby
2 swiss
```

If you want to write or update the elements, you need the indices. A common way to do that is to combine the functions `range` and `len`:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
    print(i + 1)
```

This loop traverses the list and updates each element. `len` returns the number of elements in the list. range returns a list of indices from 0 to n − 1, where n is the length of the list. Each time through the loop, i gets the index of the next element. The assignment statement in the body uses i to read the old value of the element and to assign the new value.

Nested List: Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
["spam", 1, ["Brie", "Roquefort", "Pol le Veq"], [1, 2, 3]]
```

While Loop and Lists

```
a_list = ["a", "b", "c"]
print(a_list)

count = 0
while count <= 5:
  print(count)
  a_list.append(count)
  count = count + 1
#  count += 1

print(a_list)
```

The code above begins by creating a list named **a_list**. The list is initially populated with the strings "a", "b", and "c". Then that list is printed producing the first line of output text shown in the next figure.

Create and Initialize a Counter Variable

Following that, a variable named count is created and initialized with the value 0. This variable will be used to control the number of iterations in the while loop.

Execute a While Loop

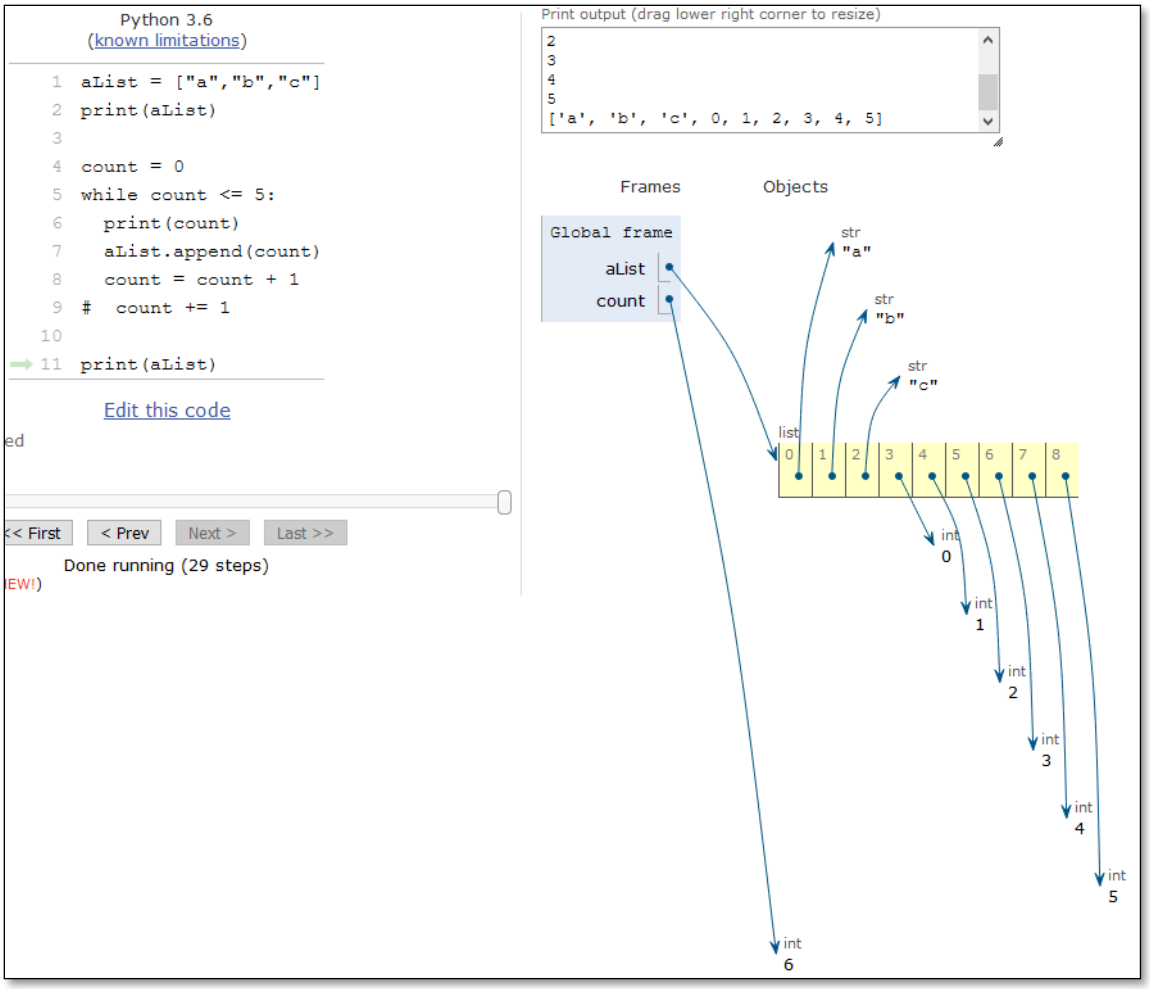
The next four lines of code in Listing 2 constitute a while loop. The first line in the while loop contains a decision structure terminated by a colon.

Recall that the initial value of count is 0. The behavior of the first line in the while loop can be paraphrased as follows:

> While the value of count is less than or equal (note the relational operator that consists of a left angle bracket and an equal character) to the literal value 5, execute all the statements that follow the colon at the same indentation level.

Stated differently, for as long as the conditional clause (count less than or equal to 5) continues to be true, execute all the statements that follow the colon at the same indentation level. When the conditional clause is no longer true, skip the indented statements and transfer control to whatever follows the indented statements.

In this case, there are three statements (followed by a comment) at the same indentation level following the colon. The first of the three statements prints the current value stored in the variable named count as shown by the second line of text in the visualization.



Tutorial 6.1 – A List of Numbers

Add numbers to a list from user input and display the list. This tutorial demonstrates two ways to use a for loop to go through a list one item at a time.

```python
"""
    Name: number_list.py
    Author:
    Created:
    Purpose: Enter numbers into a list and display
"""
# The size or length of the list
NUMBER_OF_ENTRIES = 5


def main():
    # Create list to hold numbers
    numbers = [0] * NUMBER_OF_ENTRIES

    # Prompt the user for input
    print(f"Please enter {NUMBER_OF_ENTRIES} numbers: ")

    # Loop through the list one index at a time
    # len gives the length of the list
    for i in range(len(numbers)):

        # Get input from user, place in the list
        numbers[i] = float(input(f"Enter number {i + 1}: "))

    print("You entered the following numbers.")

    # Loop through the list one item value at a time
    for value in numbers:
        print(value, end=" ")


# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example run:

```
Please enter 5 numbers:
Enter number 1: 122
Enter number 2: 123
Enter number 3: 124
Enter number 4: 125
Enter number 5: 126
You entered the following numbers.
122.0
123.0
124.0
125.0
126.0
```

List Operations

Many list operations work the same as string operations.

Expression	Result
`[7,8]+[3,4,5]`	`[7,8,3,4,5]`
`[7,8]*3`	`[7,8,7,8,7,8]`
`[0]*5`	`[0,0,0,0,0]`

The + operator concatenates lists:

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print(c)

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

The ***** operator repeats a list a given number of times:

```
alist = [0]
alist = alist * 4
print(alist)
[0, 0, 0, 0]
blist = [1, 2, 3]
blist = blist * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

The first example repeats four times. The second example repeats the list three times.

List Functions

There are many built-in functions that operate on lists without writing your own loops.

Function	Description
`len`	Returns the number of items in the list
`sum`	Returns the sum of the items in the list
`min`	Returns the minimum of the items in the list
`max`	Returns the maximum of the items in the list

Examples

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print(len(nums))
```

```
6
>>> print(max(nums))
74
>>> print(min(nums))
3
>>> print(sum(nums))
154
>>> print(sum(nums)/len(nums))
25
```

The `sum()` function only works when the list elements are numbers. The other functions (`max()`, `len()`, etc.) work with lists of strings and other types that can be comparable.

Here is a program that computes the average of a list of numbers entered by the user using a list.

First, the program to compute an average without a list:

```
total = 0
count = 0
while (True):
    inp = input("Enter a number: ")
    if inp == "done":
        break
    value = float(inp)
    total = total + value
    count = count + 1

average = total / count
print(f"Average: {average}”)
```

In this program, we have count and total variables to keep the number and running total of the user’s numbers as we repeatedly prompt the user for a number.

We could simply remember each number as the user entered it and use built-in functions to compute the sum and count at the end.

```
numlist = list()
while (True):
    inp = input("Enter a number: ")
    if inp == "done":
        break
    value = float(inp)
    numlist.append(value)
average = sum(numlist) / len(numlist)
print(f"Average: {average}")
```

We make an empty list before the loop starts, and then each time we have a number, we append it to the list. At the end of the program, we simply compute the sum of the numbers in the list and divide it by the count of the numbers in the list to come up with the average.

in

The in operator tells you if a list contains something.

```
if 2 in L:
   print("Your list contains the number 2. ")
if 0 not in L:
   print("Your list has no zeroes.")
```

List Methods

Python provides methods that operate on lists. Methods use the **.** (dot) operator.

Method	Description
`append(x)`	adds x to the end of the list
`sort()`	sorts the list
`count(x)`	returns the number of times x occurs in the list
`index(x)`	returns the location of the first occurrence of x
`reverse()`	reverses the list
`remove(x)`	removes first occurrence of x from the list
`pop(p)`	removes the item at index p and returns its value
`insert(p,x)`	inserts x at index p of the list

Important note: There is a big difference between list methods and string methods: String methods do not change the original string, but list methods do change the original list. To sort a list L, use `L.sort()` and not `L = L.sort()`.

wrong	*right*
`s.replace('X','x')`	`s = s.replace('X','x')`
`L = L.sort()`	`L.sort()`

append()

Append adds a new element to the end of a list:

```
t = ["a", "b", "c"]
t.append('d')
print(t)

['a', 'b', 'c', 'd']
```

insert()

Insert places a new element at the designate index and moves the others aside.

```
t = ["a", "b", "c"]
t.insert(1, "d")
print(t)
['a', 'd', 'b', 'c']
```

sort()

Arranges the elements of the list from low to high:

```
t = ["d", "c", "e", "b", "a"]
t.sort()
print(t)
['a', 'b', 'c', 'd', 'e']
```

Most list methods are void; they modify the list and return None. If you accidentally write `t = t.sort()`, you will be disappointed with the result.

reverse()

The reverse() reverses the original order of the list. It doesn't sort backward alphabetically, it reverses the current order of the list,

```
t = ["d", "c", "e", "b", "a"]
t.reverse()
print(t)
t.sort()
print(t)
['d', 'c', 'e', 'b', 'a']
['a', 'b', 'c', 'd', 'e']
```

index()

Returns the index of the first occurrence on the list of the element that is given to index() as argument. A runtime error is generated if the element is not found on the list.

```
fruitlist = ["apple", "banana", "cherry", "banana", "durian"]
print(fruitlist.index("banana"))
1
```

count()

Returns an integer that indicates how often the element that is passed to it as an argument occurs in the list.

```
fruitlist = ["apple", "banana", "cherry", "banana", "durian"]
print(fruitlist.count("banana"))
2
```

remove()

Removes an item with a certain value.

```
fruitlist = ["apple", "banana", "cherry", "banana", "durian"]
fruitlist.remove("banana")
print(fruitlist)
['apple', 'cherry', 'banana', 'durian']
```

del

`del` is a function, not a method. `del` removes an item at an index.

```
t = ["a", "b", "c"]
del t[1]
print(t)
['a', 'c']
```

pop()

`pop` removes an item from a list at a designated index and returns a value. You can assign that value to a variable. If you don't provide an index, it deletes and returns the last element.

```
fruitlist = ["apple", "banana", "cherry", "banana", "durian"]
popfruit = fruitlist.pop(1)
print(fruitlist)
print(popfruit)
['apple', 'cherry', 'banana', 'durian']
Banana
```

extend()

Takes a list as an argument and appends all the elements. This example leaves t2 unmodified.

```
t1 = ["a", "b", "c"]
t2 = ["d", "e"]
t1.extend(t2)
print(t1)
['a', 'b', 'c', 'd', 'e']
```

Tutorial 6.2 – Working with Lists

The following program manipulates a list using various methods and functions.

Name the file: **list_manipulation.py**

```python
"""
    Name: list_manipulation.py
    Author:
    Created:
    Purpose: Demonstrate various list methods and functions
"""


def main():
    # Create and populate the list
    list = [4, -4, 7, 9, 10, 100]
    print(f"Original list:  {list}")

    # Get the length of the list
    print(f"Length of list: {len(list)}")

    # Sort the list
    list.sort()
    print(f"Sorted List: {list}")

    # Reverse the lilst
    list.reverse()
    print(f"Reversed list: {list}")

    # Remove the element value 9 from the list
    list.remove(9)
    print(f"Removed 9 from list: {list}")

    # Add 50 to the end of the list
    list.append(50)
    print(f"Appended 50 to end of list: {list}")

    # Sort the list
    list.sort()
    print(f"Sorted List: {list}")


"""
    If a standalone program, call the main function
    Else, use as a module
"""
if __name__ == "__main__":
    main()
```

Example run:

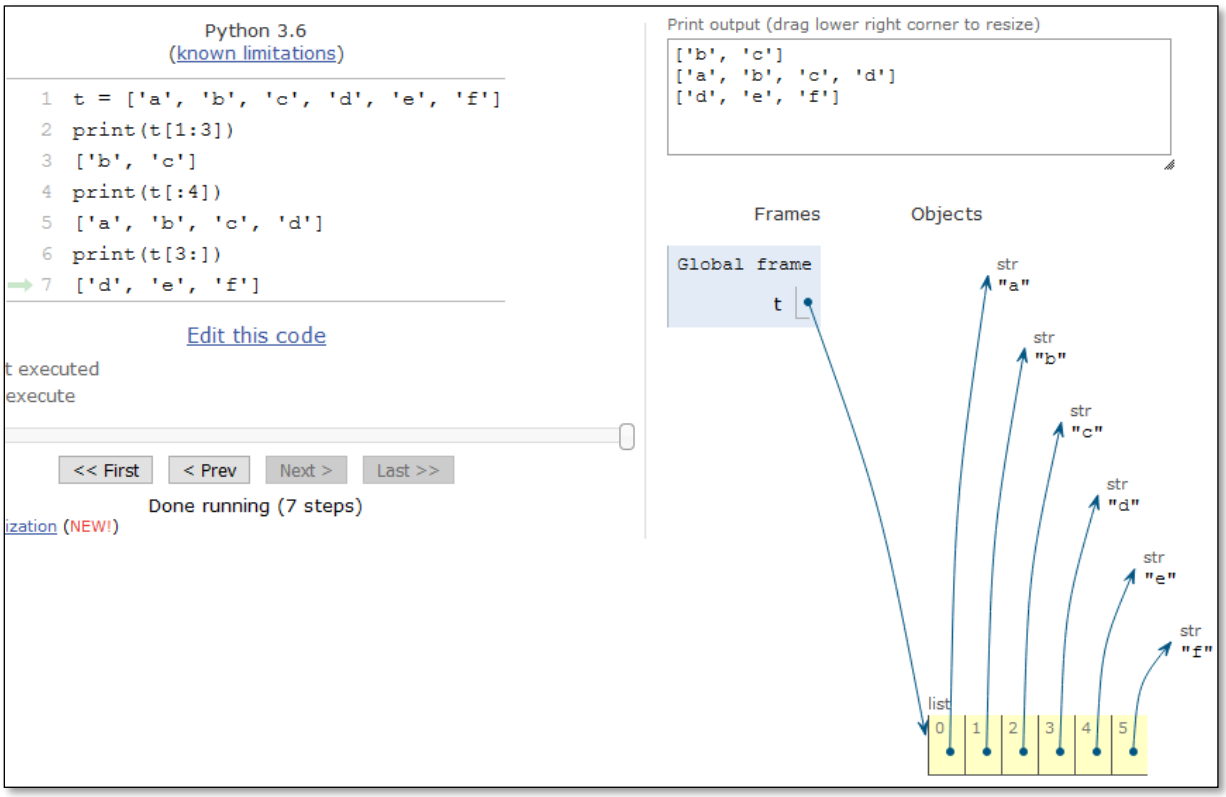
```
Original list:  [4, -4, 7, 9, 10, 100]
Length of list:  6
Sorted List:  [-4, 4, 7, 9, 10, 100]
Reversed list:  [100, 10, 9, 7, 4, -4]
Removed 9 from list:   [100, 10, 7, 4, -4]
Appended 50 to end of list:  [100, 10, 7, 4, -4, 50]
Sorted List:  [-4, 4, 7, 10, 50, 100]
```

Slice

The **slice** operator works on lists. The start is the index of the first element of the slice. The end is the index denoting the end of the slice. The end is not inclusive.

`list_name[start : end]`

```
t = ["a", "b", "c", "d", "e", "f"]
print(t[1:3])
['b', 'c']
print(t[:4])
['a', 'b', 'c', 'd']
print(t[3:])
['d', 'e', 'f']
```



`t[1:3]` slices the index 1 and 2 items.

`t[:4]` slices everything from the beginning of the list to index 3.

`t[3:]` slices everything after and including index 3.

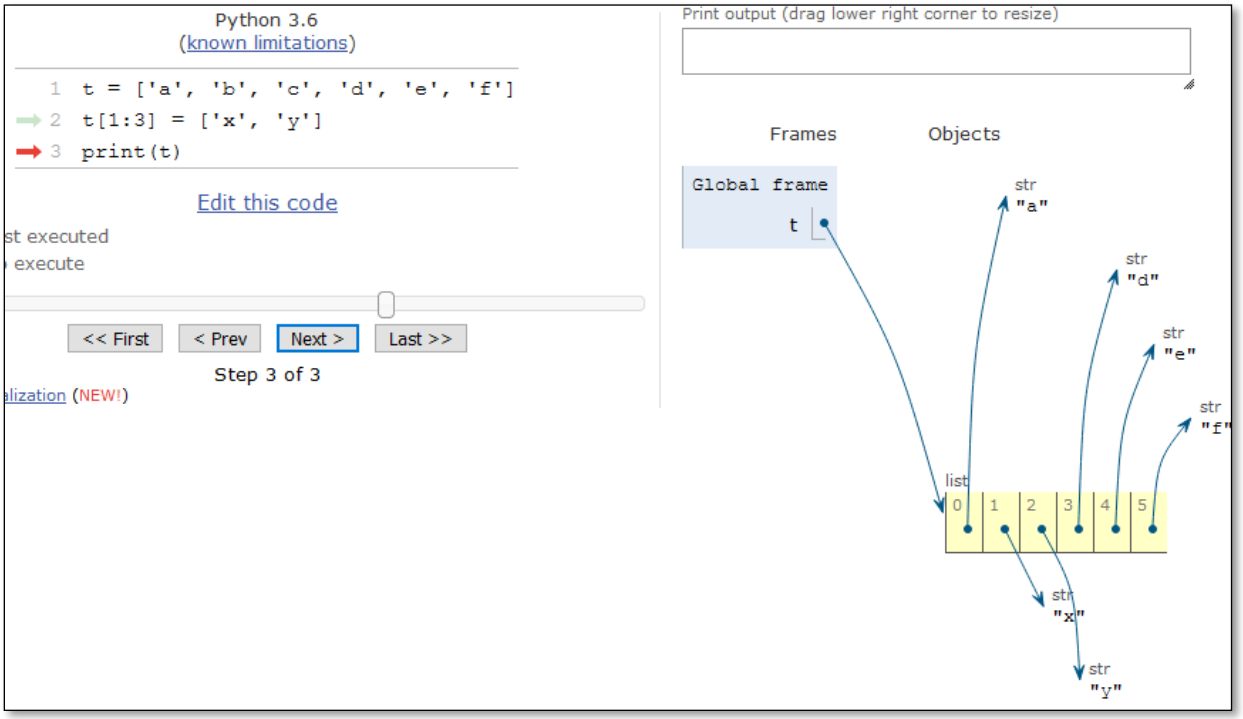
If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. If you omit both, the slice is a copy of the whole list.

```
print(t[:])
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that fold, spindle, or mutilate lists.

A slice operator on the left side of an assignment can update multiple elements:

```
t = ["a", "b", "c", "d", "e", "f"]
t[1:3] = ['x', 'y']
print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```



To remove more than one element, you can use `del` with a `slice` index:

```
t = ["a", "b", "c", "d", "e", "f"]
del t[1:5]
print(t)
['a', 'f']
```

`slice` selects all the elements up to, but not including, the second index.

Lists and Strings

A string is a sequence of characters. A list is a sequence of values. A list of characters is not the same as a string.

list()

To convert from a string to a list of characters, you can use **list**:

```
s = "spam"
t = list(s)
print(t)

['s', 'p', 'a', 'm']
```

Because `list` is the name of a built-in function, you should avoid using it as a variable name. Avoid the letter "l" because it looks too much like the number "1". You might want to use "t".

split()

The `list` function breaks a string into individual letters. If you want to break a string into words, you can use the `split` method:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print(t)
['pining', 'for', 'the', 'fjords']
>>> print(t[2])
the
```

Once you have used split to break the string into a list of words, you can use the index operator (square bracket) to look at a word in the list.

You can call split with an optional argument called a delimiter that specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

join()

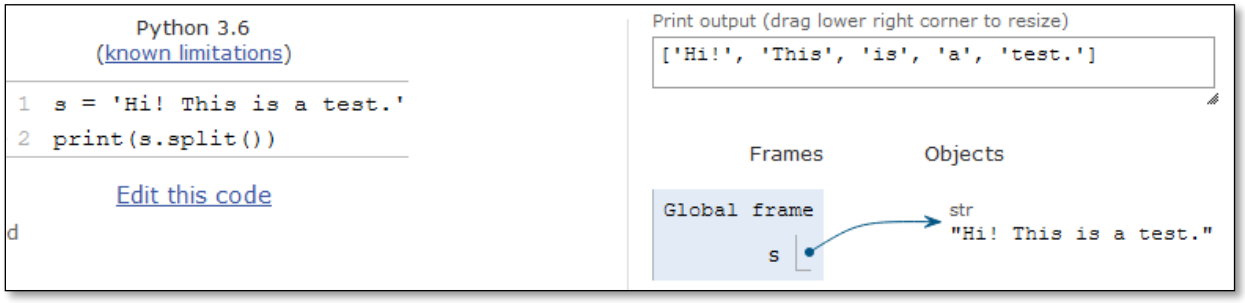
join is the inverse of split. It takes a list of strings and concatenates the elements. join is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

In this case the delimiter is a space character, so join puts a space between words. To concatenate strings without spaces, you can use the empty string, "", as a delimiter.

The split method returns a list of the words of a string. The method assumes that words are separated by whitespace, which can be either spaces, tabs or newline characters. Here is an example:

```
s = "Hi! This is a test. "
print(s.split())
['Hi!', 'This', 'is', 'a', 'test.']
```



Since `split` breaks up the string at spaces, the punctuation will be part of the words. There is a module called `string` that contains, among other things, a string variable called `punctuation` that contains common punctuation. We can remove the punctuation from a string `s` with the following code:

```
from string import punctuation
s = "cha,ra.c;ter"
for c in punctuation:
   s = s.replace(c, '')
print(s)
character
```

Here is a program that counts how many times a certain word occurs in a string.

```
from string import punctuation
s = input("Enter a string: ")
L = []
for c in punctuation:
   s = s.replace(c, "")
   s = s.lower()
   L = s.split()
word = input("Enter a word: ")
print(f"{word} appears {L.count(word)} times.")
```

Example run:

```
Enter a string: eric, idle, eric..
Enter a word: eric
eric appears 2 times.
```

Optional Argument: The `split` method takes an optional argument that allows it to break the string at places other than spaces. Here is an example:

```
s = "1-800-271-8281"
print(s.split("-"))
['1', '800', '271', '8281']
```

Tutorial 6.3 – Input a Contact List

Let's use the append and sort functions to add items to a list. Create a program named **list_append.py**

```python
"""
    Name: list_append.py
    Author:
    Created:
    Purpose: Create a list from user input
"""


def main():
    # Create an empty list.
    friends = []

    # Add names to the list.
    while True:
        # Get a name from the user.
        name = input("Enter a name (Press Enter to quit): ")

        # Exit the loop if the user presses enter
        if name == "":
            break

        # Append the name to the list.
        friends.append(name)

    # Print a blank line
    print()

    # Sort the names in the list
    friends.sort()

    # Inform the user you will print the list
    print("Here is your sorted contact list:")

    # Iterate through and print your list
    for name in friends:
        print(name)


# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example run:

```
Enter a name (Press Enter to quit): Alvin
Enter a name (Press Enter to quit): Theadore
Enter a name (Press Enter to quit): Brian
Enter a name (Press Enter to quit):

Here is your sorted contact list:
Alvin
Brian
Theadore
```

Tutorial 6.4 – Shopping List

The following program demonstrates how to create and manipulate a list.

Create a Python program called **shopping_list.py**.

```
"""
    Name: shopping_list.py
    Author:
    Created:
    Purpose: A demonstration shopping list
"""

# Constant for list size
LIST_SIZE = 5

def main():
    # Create an empty list for 5 items
    shoplist = [0] * LIST_SIZE

    print(f"Please enter {LIST_SIZE} items for your grocery list.")

    # Iterate through the list prompting for user input
    for index in range(len(shoplist)):
        shoplist[index] = input(f"Item #{index + 1}: ")

    print(f"\nMy grocery list is: ", end="")

    # Iterate through and print the shopping list
    for item in shoplist:
        print(item, end=" ")

    # Add an item to the list
    print("\n\nI also have to buy Rice.")
    shoplist.append("Rice")
    print(f"My shopping list is now {shoplist}")

    print("\nI will sort my list now")

    # Sort the list
    shoplist.sort()
    print(f"Sorted shopping list is {shoplist}")

    # Remove item from list
    print(f"\nThe first item I will buy is {shoplist[0]}")
    olditem = shoplist[0]
    del shoplist[0]
    print(f"I bought the {olditem}")

    # Final list
    print(f"\nMy shopping list is now {shoplist}")

# If a standalone program, call the main function
# Else, use as a module
if __name__ == "__main__":
    main()
```

Example run:

```
Please enter 5 items for your grocery list.
Item #1: Salmon
Item #2: Onions
Item #3: Hamburger
Item #4: Buns
Item #5: Ketchup

My grocery list is:  Salmon Onions Hamburger Buns Ketchup

I also have to buy Rice.
My shopping list is now ['Salmon', 'Onions', 'Hamburger', 'Buns', 'Ketchup', 'Rice']

I will sort my list now
Sorted shopping list is ['Buns', 'Hamburger', 'Ketchup', 'Onions', 'Rice', 'Salmon']

The first item I will buy is Buns
I bought the Buns

My shopping list is now ['Hamburger', 'Ketchup', 'Onions', 'Rice', 'Salmon']
```

Parsing Lines

Usually when we are reading a file, we want to do something to the lines other than just printing the whole line. Often, we want to find the "interesting lines" and then parse the line to find some interesting part of the line. What if we wanted to print out the day of the week from those lines that start with "From"?

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

The split method is very effective when faced with this kind of problem. We can write a small program that looks for lines where the line starts with "From", split those lines, and then print out the third word in the line:

```
fhand = open("mbox-short.txt")
for line in fhand:
line = line.rstrip()
if not line.startswith('From '): continue
words = line.split()
print(words[2])
```

The program produces the following output:

```
Sat
Fri
Fri
Fri
...
```

Lists and the Random Module

There are some nice functions in the random module that work on lists.

Function	Description
`choice(L)`	picks a random item from `L`
`sample(L, n)`	picks a group of `n` random items from `L`
`shuffle(L)`	Shuffles the items of `L`

Note: The shuffle function modifies the original list. If you don't want your list changed, you'll need to make a copy of it.

Choice

This can be used to pick a name from a list of names.

```
from random import choice
names = ["Joe", "Bob", "Sue", "Sally"]
current_player = choice(names)
print(current_player)
Bob
```

This function also works with strings, picking a random character from a string. Here is an example that uses choice to fill the screen with a bunch of random characters.

```
from random import choice
s='abcdefghijklmnopqrstuvwxyz1234567890!@#$%^&*()'
for i in range(10000):
    print(choice(s), end='')
```

Choice can also pick random numbers out of a list.

```
# Import the choice function from random
from random import choice
numbers = [1, 2, 3, 4]
random_choice = choice(numbers)
print(random_choice)
2
```

Choices

Choices allows picking more than one element from a list. The second argument k = 2, sets how many random elements to return.

```
# Import the choice function from random
from random import choices
```

```
numbers = [1, 2, 3, 4]
random_choices = choices(numbers, k = 2)
print(random_choices)
[2, 3]
```

Sample

This function is similar to choice. Whereas choice picks one item from a list, sample can be used to pick several.

```
from random import sample
names = ["Joe", "Bob", "Sue", "Sally"]
team = sample(names, 2)
```

Shuffle

Shuffle picks a random ordering of players in a game. The shuffle function modifies the original list. If you don't want your list changed, you'll need to make a copy of it.

Here is a nice use of shuffle to pick a random ordering of players in a game.

```
from random import shuffle
players = ["Joe", "Bob", "Sue", "Sally"]
shuffle(players)
for p in players:
    print(p, 'it is your turn.')
    # code to play the game goes here
```

Here we use shuffle divide a group of people into teams of two. Assume we are given a list called `names`.

```
from random import shuffle
names = ["Joe", "Bob", "Sue", "Sally"]
shuffle(names)
teams = []
for i in range(0, len(names),2):
   teams.append([names[i], names[i+1]])
```

Each item in teams is a list of two names. The way the code works is we shuffle the names so they are in a random order. The first two names in the shuffled list become the first team, the next two names become the second team, etc. Notice that we use the optional third argument to range to skip ahead by two through the list of names.

List Arguments

When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change.

For example, delete_head removes the first element from a list:

```
def delete_head(t):
   del t[0]
```

Here's how it is used:

```
def delete_head(t):
   del t[0]
letters = ["a", "b", "c"]
delete_head(letters)
print(letters)
['b', 'c']
```

The parameter t and the variable letters are aliases for the same object.

It is important to distinguish between operations that modify lists and operations that create new lists.

For example, the append method modifies a list, but the + operator creates a new list:

```
t1 = [1, 2]
t2 = t1.append(3)

print(t1)
[1, 2, 3]

print(t2)
None
t3 = t1 + [3]

print(t3)
[1, 2, 3]

print(t2 is t3)
False
```

This difference is important when you write functions that are supposed to modify lists.

For example, tail returns all but the first element of a list:

```
def tail(t):
  return t[1:]
```

This function leaves the original list unmodified. Here's how it is used:

```
def tail(t):
  return t[1:]
letters = ["a", "b", "c"]
rest = tail(letters)

print(rest)
['b', 'c']
```

Two-dimensional Lists

There are several common things that can be represented by two-dimensional lists, like a Tictac-toe board or the pixels on a computer screen. In Python, one way to create a two-dimensional list is to create a list whose items are themselves lists. Here is an example:

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

Indexing: We use two indices to access individual items. To get the entry in row r, column c, use the following:

```
L[r][c]
```

Printing a two-dimensional list: To print a two-dimensional list, you can use nested for loops.

The following example prints a 10 x 5 list:

```
for r in range(10):
for c in range(5):
print(L[r][c], end=" ")
print()
```

Another option is to use the **pprint** function of the **pprint** module. This function is used to "pretty-print" its argument. Here is an example to print a list L:

```
from pprint import pprint
pprint(L)
```

The `pprint` function can be used to nicely print ordinary lists and other objects in Python.

Working with two-dimensional lists: Nested for loops, like the ones used in printing a two-dimensional list, can also be used to process the items in a two-dimensional list. Here is an example that counts how many entries in a 10 x 5 list are even.

```
count = 0
for r in range(10):
    for c in range(5):
        if L[r][c]%2==0:
            count = count + 1
```

Creating large two-dimensional lists: To create a larger list, you can use a list comprehension like below:

```
L = [[0]*50 for i in range(100)]
```

This creates a list of zeroes with 100 rows and 50 columns.

Picking out rows and columns: To get the rth row of L, use the following:

```
L[r]
```

To get the cth column of L, use a list comprehension:

```
[L[i][c] for i in range(len(L))]
```

Example 1 – Random List

Write a program that generates a list named **`random_list`** of 50 random numbers between 1 and 100.

```
from random import randint
# Create an empty list
random_list = []
# Generate and add random integers to the list
for i in range(50):
    randum_number = randint(1, 100)
    random_list.append(random_number)
    print(random_list[i])
```

Example run (Only a partial list is shown):

```
53
15
15
29
92
4
35
```

We used the append method to build up the list one item at a time starting with the empty list, [].

Example 2 – Squared List

Replace each element in a list `square_list` with its square.

```
square_list = [1, 2, 3]
for i in range(len(square_list)):
    # Use end="" to keep everything on the same line
    print(square_list[i], end="")
    square_list[i] = square_list[i]**2
    print(f" squared: {square_list[i]}")
```

Example run:

```
1 squared: 1
2 squared: 4
3 squared: 9
```

Example 3

Count how many items in a list L are greater than 50.

```
count = 0
for item in L:
   if item > 50:
      count = count + 1
```

Example 4

Given a list L that contains numbers between 1 and 10, create a new list whose first element is how many ones are in L, whose second element is how many twos are in L, etc.

```
L = [1, 2, 3, 1, 5, 5, 9, 9]
frequencies = []
for i in range(1, 10):
    frequencies.append(L.count(i))
    print(frequencies[i - 1])
```

The key is the list method count that tells how many times a something occurs in a list.

Example 5

Write a program that prints out the two largest and two smallest elements of a list called scores.

```
scores = [24, 45, 1, 6]
scores.sort()
print(f"Two smallest: {scores[0]} {scores[1]}")
print(f"Two largest: {scores[-1]} {scores[-2]} ")
```

Once we sort the list, the smallest values are at the beginning and the largest are at the end.

Example 6

Here is a program to play a simple quiz game.

```
questions = ["What is the capital of France? ",
"Which state has only one neighbor? "]
answers = ["Paris", "Maine"]
num_right = 0
for i in range(len(questions)):
    guess = input(questions[i])
    if guess.lower()==answers[i].lower():
        print("Correct")
        num_right=num_right+1
    else:
        print(f"Wrong. The answer is {answers[i]} ")
    print(f"You have {num_right} out of {i+1} right. ")
```

This illustrates the general technique: If you find yourself repeating the same code over and over, try lists and a for loop. The few parts of your repetitious code that are varying are where the list code will go.

The benefits of this are that to change a question, add a question, or change the order, only the questions and answers lists need to be changed. Also, if you want to make a change to the program, like not telling the user the correct answer, then all you have to do is modify a single line, instead of twenty copies of that line spread throughout the program.

Tuples

A tuple is like a list whose values cannot be modified. It is an ordered list of objects, and it can contain references to any type of object. A Python tuple is called an array in other programming languages.

- Tuples are normally written as a sequence of items contained in (optional) matching parentheses.
- A tuple is an immutable sequence.
- Items in a tuple are accessed using a numeric index.
- Tuples can be nested.

A tuple is like a list whose values cannot be modified. In other words, a tuple is immutable.

According to Lutz and Ascher, Learning Python from O'Reilly, tuples are "Ordered collections of arbitrary objects. They work exactly like lists, except that tuples can't be changed in place (they're immutable)..."

Unlike lists, however, tuples don't use square brackets for containment. They are normally written as a sequence of items contained in parentheses.

Like a string or a list, a tuple is a sequence. Like a string (but unlike a list), a tuple is an immutable sequence.

Tuples can contain other compound objects, including lists, dictionaries, and other tuples. Hence, tuples can be nested.

A tuple is a sequence of values much like a list. The values stored in a tuple can be any type, and they are indexed by integers. The important difference is that tuples are immutable. Tuples are also comparable and hashable so we can sort lists of them and use tuples as key values in Python dictionaries.

Processing a tuple is faster than a list. They also require less memory. Tuples are safe as they can’t be changed. There are some operations in Python that require the use of a tuple.

Why do Tuples Exist?

Tuples provide some degree of integrity to the data stored in them. You can pass a tuple around through a program and be confident that its value can't be accidentally changed. As mentioned earlier, however, the values stored in the items referred to in a tuple can be changed.

A tuple is a comma-separated list of values:

```
t = ("a", "b", "c", "d", "e")
```

Most list operators also work on tuples. The ones that modify the list itself will not work. The bracket operator indexes an element:

```
t = ("a", "b", "c", "d", "e")
print(t[0])
a
```

The slice operator selects a range of elements.

```
t = ("a", "b", "c", "d", "e")
print(t[1:3])
('b', 'c')
```

You can't modify the elements of a tuple, but you can replace one tuple with another:

```
t = ("a", "b", "c", "d", "e")
t = ("A",) + t[1:]
print(t)
('A', 'b', 'c', 'd', 'e')
```

Dictionaries

A dictionary is a more general version of a list. Dictionaries are mutable. Here is a list that contains the number of days in the months of the year:

```
days = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

If we want the number of days in January, use `days[0]`. December is `days[11]` or `days[-1]`.

Here is a dictionary of the days in the months of the year:

```
days = {"January":31, "February":28, "March":31, "April":30, "May":31,
       "June":30, "July":31, "August":31, "September":30, "October":31,
       "November":30, "December":31}
```

To get the number of days in January, we use `days["January"]`. One benefit of using dictionaries here is the code is more readable, and we don't have to figure out which index in the list a given month is at.

Unlike other sequences, items in dictionaries are unordered. The first item in a list named spam would be spam[0]. There is no "first" item in a dictionary. The dictionary does

remember the order of entry. While the order of items matters for determining whether two lists or tuples are the same, it does not matter in what order the key-value pairs are typed in a dictionary.

Dictionary Basics

Parameter	Details
`key`	The desired key to lookup
`value`	The value to set or return

Literal Syntax

```
d = {} # empty dict
d = {"key": "value"} # dict with initial values
```

Creating Dictionaries

Here is a simple dictionary:

```
d = {"A":100, "B":200}
```

To declare a dictionary, we enclose it in curly braces, `{}`. Each entry consists of a pair separated by a colon. The first part of the pair is called the key and the second is the value. The key acts like an index. In the first pair, "A":100, the key is "A", the value is 100, and d["A"] gives 100.

Keys are often strings. They can be integers, floats, and many other things as well. You can mix different types of keys in the same dictionary and different types of values, too.

Changing Dictionaries

Let's start with this dictionary:

```
d = {"A":100, "B":200}
```

To change `d["A"]` to 400, do

```
d["A"] = 400
```

To add a new entry to the dictionary, we can just assign it, like below:

```
d["C"] = 500
```

Note that this sort of thing does not work with lists. Doing `L[2]=500` on a list with two elements would produce an index out of range error. But it does work with dictionaries.

To delete an entry from a dictionary, use the `del` operator:

```
del d["A"]
```

Empty dictionary: The empty dictionary is `{}`, which is the dictionary equivalent of `[]` for lists or "" for strings.

Tutorial 6.5 – Create a Dictionary

You can use a dictionary as an actual dictionary of definitions. Create the following program and save it as **cats_and_dogs.py**

```
1  # Name: cats_and_dogs.py
2  # Author:
3  # Created:
4  # Purpose: Create and use a dictionary
5  # Create the dictionary
6
7  def main():
8      # Define the key : value pairs of the dictionary
9      dictionary = {'dog': 'has a tail and goes woof!',
10                   'cat': 'says meow',
11                   'mouse': 'is chased by cats'}
12
13     # Prompt the user to enter a dictionary key
14     print("This dictionary contains values for dog, cat, or mouse.")
15     word = input('Enter a word (key): ')
16
17     # Use the key entered by the user to access the value
18     print(f'The value associated with {word}: {dictionary[word]}')
19     print(f'A {word} {dictionary[word]}')
20
21 # Call the main function.
22 if __name__ == '__main__':
23     main()
```

Example run:

```
This dictionary contains values for dog, cat, or mouse.
Enter a word (key): mouse
The value associated with mouse: is chased by cats
A mouse is chased by cats
```

Dictionary Examples

Example 1 The following dictionary is useful in a program that works with Roman numerals.

```
numerals = {'I':1, 'V':5, 'X':10, 'L':50, 'C':100, 'D':500, 'M':1000}
```

Example 2 In the game Scrabble, each letter has a point value associated with it. We can use the following dictionary for the letter values:

```
points = {'A':1, 'B':3, 'C':3, 'D':2, 'E':1, 'F':4, 'G':2,
          'H':4, 'I':1, 'J':8, 'K':5, 'L':1, 'M':3, 'N':1,
          'O':1, 'P':3, 'Q':10, 'R':1, 'S':1, 'T':1, 'U':1,
          'V':4, 'W':4, 'X':8, 'Y':4, 'Z':10}
```

To score a word, we can do the following:

```
score = sum([points[c] for c in word])
```

Or, if you prefer the long way:

```
total = 0
for c in word:
    total += points[c]
```

Example 3 A dictionary provides a nice way to represent a deck of cards:

```
deck = [{'value':i, 'suit':c}
for c in ['spades', 'clubs', 'hearts', 'diamonds']
    for i in range(2, 15)]
```

The deck is a list of 52 dictionaries. The shuffle method can be used to shuffle the deck:

```
shuffle(deck)
```

The first card in the deck is `deck[0]`. To get the value and the suit of the card, we would use the following:

```
deck[0]['value']
deck[0]['suit']
```

Working with Dictionaries

Accessing a Dictionary by Index: You can access a dictionary by an index number just like a list by temporarily converting the dictionary to a list.

```
dict = {'A':100, 'B':200}
print(list(dict)[0])
A
```

Copying dictionaries: Just like for lists, making copies of dictionaries is a little tricky. To copy a dictionary, use its `copy` method. Here is an example:

```
d2 = d.copy()
```

in The `in` operator is used to tell if something is a key in the dictionary. For instance, say we have the following dictionary:

```
d = {'A':100, 'B':200}
```

Referring to a key that is not in the dictionary will produce an error. For instance, `print(d['C'])` will fail. To prevent this error, we can use the `in` operator to check first if a key is in the dictionary before trying to use the key. Here is an example:

```
letter = input('Enter a letter: ')
if letter in d:
    print('The value is', d[letter])
else:
    print('Not in dictionary')
```

You can also use `not in` to see if a key is not in the dictionary.

Looping: Looping through dictionaries is like looping through lists. Here is an example that prints the keys in a dictionary:

```
for key in d:
    print(key)
```

Here is an example that prints the values:

```
for key in d:
    print(d[key])
```

Lists of keys and values: The following table illustrates the ways to get lists of keys and values from a dictionary. It uses the dictionary `d={'A':1,'B':3}`.

Statement	Result	Description
`list(d)`	`['A', 'B']`	keys of d
`list(d.values())`	`[1, 3]`	values of d
`list(d.items())`	`[('A',1),('B',3)]`	(key,value) pairs of d

The pairs returned by `d.items` are tuples.

Here is a use of `d.items` to find all the keys in a dictionary `d` that correspond to a value of 100:

```
d = {'A':100, 'B':200, 'C':100}
L = [x[0] for x in d.items() if x[1]==100]
['A', 'C']
```

dict The `dict` function is another way to create a dictionary. One use for it is kind of like the opposite of the `items` method:

```
d = dict([('A', 100),('B', 300)])
```

This creates the dictionary `{'A':100, 'B':300}`. This way of building a dictionary is useful if your program needs to construct a dictionary while it is running.

Iterating over Dictionaries

We will be looping over the keys in the dictionary, not the values.

```
dict = {
    "c": 10,
    "b": 20,
    "a": 30
}
for key in dict:
    print(f"key {key} has value {dict[key]}")
```

This gives us the output:

```
key c has value 10
key b has value 20
key a has value 30
```

In the latest version of Python, the keys come out in the same order they've been added to the dictionary.

If you want them in another order, there are options:

```
dict = {
    "c": 10,
    "b": 20,
    "a": 30
}
for key in sorted(dict): # <--- Add the call to sorted()
    print(f"key {key} has value {dict[key]}")
```

Now we have this, where the keys have been sorted alphabetically.

```
key a has value 10
key b has value 20
key c has value 30
```

Also, similar to how lists work, you can use the `.items()` method to get all keys and values out at the same time in a for loop, like this:

```
dict = {
    "c": 10,
    "b": 20,
    "a": 30
}
for key, value in dict.items():
    print(f"key {key} has value {value}")
```

Dictionary Functionality

You have several tools in your toolkit for working with dicts in Python:

Method	Description
`.clear()`	Empty a dictionary, removing all keys/values
`.copy()`	Return a copy of a dictionary
`get(key)`	Get a value from a dictionary, with a default if it doesn't exist
`.items`	() Return a list-ish of the (key,value) pairs in the dictionary
`.keys()`	Return a list-ish of the keys in the dictionary
`.values()`	Return a list-ish of the values in the dictionary
`.pop(key)`	Return the value for the given key, and remove it from the dictionary
`.popitem()`	Pop and return the most-recently-added (key,value) pair

You can get an item out of a dictionary using `the` **`.get()`** method. This method returns a default value of None if the key doesn't exist.

```
val = d.get("x")
if val is None: # Note: use "is", not "==" with "None"!
   print("Key x does not exist")
else:
   print(f"Value of key x is {d[x]}")
```

If you want to see just all the keys or values, you can get an iterable back with the `.keys()` and `.values()`:

```
dict = {
    "c": 10,
    "b": 20,
    "a": 30
}
for key in dict.keys():
    print(key) # Prints "c", "b", "a"
for value in dict.values():
    print(value) # Prints 10, 20, 30
```

Tutorial 6.6 – Address Book

A dictionary is like an address-book where you can find the address or contact details of a person by knowing only his/her name i.e. we associate keys (name) with values (details). Note that the key must be unique just like you cannot find out the correct information if you have two persons with the exact same name.

Note that you can use only immutable objects (like strings) for the keys of a dictionary but you can use either immutable or mutable objects for the values of the dictionary. This basically translates to say that you should use only simple objects for keys.

Pairs of keys and values are specified in a dictionary by using the notation:

```
d = {key1: value1, key2: value2}
```

Notice that the key-value pairs are separated by a colon and the pairs are separated themselves by commas and all this is enclosed in a pair of curly braces.

Remember that key-value pairs in a dictionary are not ordered in any manner. If you want a particular order, then you will have to sort them yourself before using it.

The dictionaries that you will be using are instances/objects of the `dict` class.

```python
"""
    Name: address_book_dictionary
    Author:
    Created:
    Purpose: Create and work with a dictionary address book
"""


def main():

    # Define the address book
    address_book = {
        "Fred": "fred@hotmail.com",
        "Larry": "larry@wall.org",
        "Mike": "mike@mike.org",
        "Spammer": "spammer@hotmail.com"
    }

    # Print out the address book
    print_address_book(address_book)

    # Print a specific address by key
    print(f"\nDisplay a specific address.")
    print(f"Fred's address is {address_book.get('Fred')}")

    # Print the number of contacts
    print(f"\nThere are {len(address_book)} contacts in the address-book")

    # Deleting a key-value pair
    print(f"Delete Spammer")
    del address_book["Spammer"]

    # Print the number of contacts
    print(f"\nThere are {len(address_book)} contacts in the address-book")

    # Print out the address book
    print_address_book(address_book)

    # Adding a key-value pair
    print(f"\nAdd Guido to the address book.")
    address_book["Guido"] = "guido@pythonstuff.org"

    # Is Guido in the address book?
    print(f"\nSee if Guido is in the address book.")
    if "Guido" in address_book:
        print(f"Guido's address is {address_book.get('Guido')}")

    print_address_book(address_book)


def print_address_book(ad_book):
    """ Print out the address book """
    print(f"\nDisplay the address book.")

    # Go through the dictionary one item at a time
    for name, address in ad_book.items():
        print(f"Contact {name} at {address}")


# Call the main function.
if __name__ == "__main__":
    main()
```

Example run:

```
Display the address book.
Contact Fred at fred@hotmail.com
Contact Larry at larry@wall.org
Contact Mike at mike@mike.org
Contact Spammer at spammer@hotmail.com

Display a specific address.
Fred's address is fred@hotmail.com

There are 4 contacts in the address-book
Delete Spammer

There are 3 contacts in the address-book

Display the address book.
Contact Fred at fred@hotmail.com
Contact Larry at larry@wall.org
Contact Mike at mike@mike.org

Add Guido to the address book.

See if Guido is in the address book.
Guido's address is guido@pythonstuff.org

Display the address book.
Contact Fred at fred@hotmail.com
Contact Larry at larry@wall.org
Contact Mike at mike@mike.org
Contact Guido at guido@pythonstuff.org
```

How It Works

We create the dictionary `ab` using the notation already discussed. We then access key-value pairs by specifying the key using the indexing operator as discussed in the context of lists and tuples. Observe the simple syntax.

We can delete key-value pairs using our old friend - the `del` statement. We simply specify the dictionary and the indexing operator for the key to be removed and pass it to the `del` statement. There is no need to know the value corresponding to the key for this operation.

Next, we access each key-value pair of the dictionary using the items method of the dictionary which returns a list of tuples where each tuple contains a pair of items - the key followed by the value. We retrieve this pair and assign it to the variables name and address correspondingly for each pair using the for in loop and then print these values in the for block.

We can add new key-value pairs by simply using the indexing operator to access a key and assign that value, as we have done for Guido in the above case.

We can check if a key-value pair exists using the `in` operator.

Glossary

aliasing A circumstance where two or more variables refer to the same object.

delimiter A character or string used to indicate where a string should be split.

element One of the values in a list (or other sequence); also called items.

equivalent Having the same value.

index An integer value that indicates an element in a list.

identical Being the same object (which implies equivalence).

list A sequence of values.

list traversal The sequential accessing of each element in a list.

nested list A list that is an element of another list.

object Something a variable can refer to. An object has a type and a value.

reference The association between a variable and its value.

Assignment Submission

Attach all program files to the assignment in BlackBoard.