Module 2, Lists

Q) Define List and explain with an example how individual items are accessed from the list.

A list is a value that contains multiple values in an ordered sequence. The term list value refers to the list itself (which is a value that can be stored in a variable or passed to a function like any other value), not the values inside the list value.

Just as string values are typed with quote characters to mark where the string begins and ends, a list begins with an opening square bracket and ends with a closing square bracket, []. Values inside the list are also called items. Items are separated with commas (that is, they are comma-delimited).

Example 1:

```
>>> [1, 2, 3]
[1, 2, 3]
>>> ['cat', 'bat', 'rat', 'elephant']
['cat', 'bat', 'rat', 'elephant']
>>> ['hello', 3.1415, True, None, 42]
['hello', 3.1415, True, None, 42]
```

Example 2:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam
['cat', 'bat', 'rat', 'elephant']
```

If the list ['cat', 'bat', 'rat', 'elephant'] stored in a variable named spam. The Python code spam[0] would evaluate to 'cat', and spam[1] would evaluate to 'bat', and so on. The integer inside the square brackets that follows the list is called an index. The first value in the list is at index 0, the second value is at index 1, the third value is at index 2, and so on. Figure below shows a list value assigned to spam, along with what the index expressions would evaluate to.

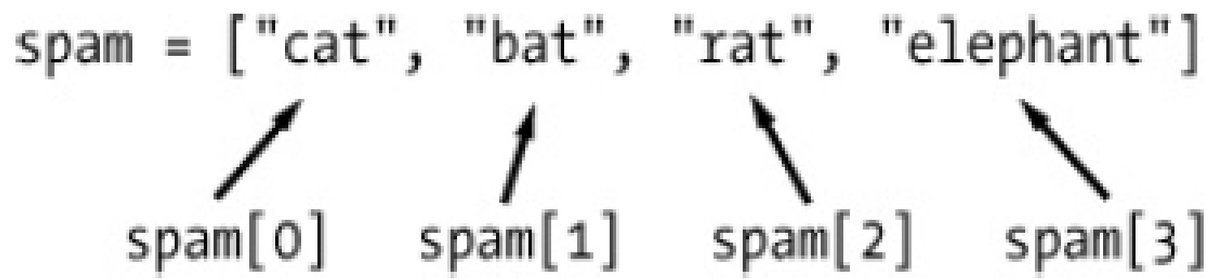

Figure. A list value stored in the variable spam, showing which value each index refers to

Example:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
```

Module 2, Lists

Q) Define List and explain with an example how individual items are accessed from the list.

A list is a value that contains multiple values in an ordered sequence. The term list value refers to the list itself (which is a value that can be stored in a variable or passed to a function like any other value), not the values inside the list value.

Just as string values are typed with quote characters to mark where the string begins and ends, a list begins with an opening square bracket and ends with a closing square bracket, []. Values inside the list are also called items. Items are separated with commas (that is, they are comma-delimited).

Example 1:

```
>>> [1, 2, 3]
[1, 2, 3]
>>> ['cat', 'bat', 'rat', 'elephant']
['cat', 'bat', 'rat', 'elephant']
>>> ['hello', 3.1415, True, None, 42]
['hello', 3.1415, True, None, 42]
```

Example 2:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam
['cat', 'bat', 'rat', 'elephant']
```

If the list ['cat', 'bat', 'rat', 'elephant'] stored in a variable named spam. The Python code spam[0] would evaluate to 'cat', and spam[1] would evaluate to 'bat', and so on. The integer inside the square brackets that follows the list is called an index. The first value in the list is at index 0, the second value is at index 1, the third value is at index 2, and so on. Figure below shows a list value assigned to spam, along with what the index expressions would evaluate to.

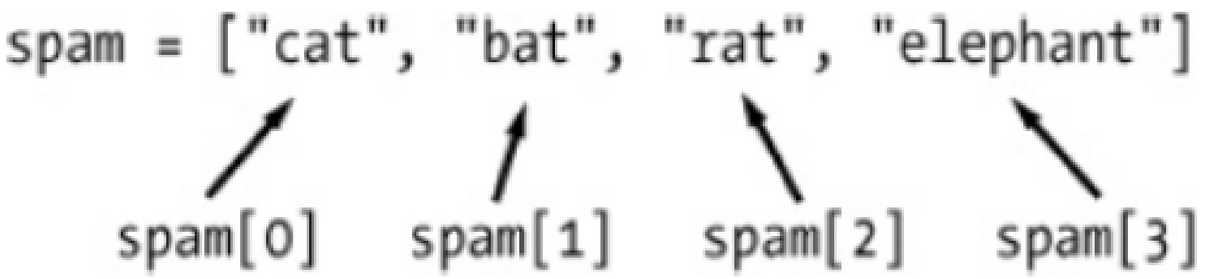

Figure. A list value stored in the variable spam, showing which value each index refers to

Example:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0]
'cat'
>>> spam[1]
'bat'
>>> spam[2]
'rat'
>>> spam[3]
'elephant'
>>> ['cat', 'bat', 'rat', 'elephant'][3]
'elephant'
```

```
>>> spam[0]
'cat'
>>> spam[1]
'bat'
>>> spam[2]
'rat'
>>> spam[3]
'elephant'
>>> ['cat', 'bat', 'rat', 'elephant'][3]
'elephant'
```

Q) Explain negative indexes and slices with suitable example.

Negative Indexes

While indexes start at 0 and go up, you can also use negative integers for the index. The integer value -1 refers to the last index in a list, the value -2 refers to the second-to-last index in a list, and so on.

Example:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[-1]
'elephant'
>>> spam[-3]
'bat'
>>> 'The ' + spam[-1] + ' is afraid of the ' + spam[-3] + '.'
'The elephant is afraid of the bat.'
```

Slices

Just as an index can get a single value from a list, a slice can get several values from a list, in the form of a new list. A slice is typed between square brackets, like an index, but it has two integers separated by a colon.

The difference between indexes and slices.

- spam[2] is a list with an index (one integer).
- spam[1:4] is a list with a slice (two integers).

In a slice, the first integer is the index where the slice starts. The second integer is the index where the slice ends. A slice goes up to, but will not include, the value at the second index. A slice evaluates to a new list value.

Example:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0:4]
['cat', 'bat', 'rat', 'elephant']
>>> spam[1:3] ['bat', 'rat']
```

```
>>> spam[0:-1]
['cat', 'bat', 'rat']
```

As a shortcut, we can leave out one or both of the indexes on either side of the colon in the slice. Leaving out the first index is the same as using 0, or the beginning of the list. Leaving out the second index is the same as using the length of the list, which will slice to the end of the list.

Example:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[:2]
['cat', 'bat']
>>> spam[1:]
['bat', 'rat', 'elephant']
>>> spam[:]
['cat', 'bat', 'rat', 'elephant']
```

Q) Explain how list value can be changed with indexes and give example.

Normally a variable name goes on the left side of an assignment statement, like spam = 42. We can also use an index of a list to change the value at that index.

Example:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
 >>> spam[1] = 'aardvark'
>>> spam ['cat', 'aardvark', 'rat', 'elephant']
>>> spam[2] = spam[1]
>>> spam ['cat', 'aardvark', 'aardvark', 'elephant']
>>> spam[-1] = 12345
>>> spam
['cat', 'aardvark', 'aardvark', 12345]
```

Q) Explain List Concatenation and List Replication with an example.

The + operator can combine two lists to create a new list value in the same way it combines two strings into a new string value.

Example:

```
 >>> [1, 2, 3] + ['A', 'B', 'C']
[1, 2, 3, 'A', 'B', 'C']
>>> spam = [1, 2, 3]
>>> spam = spam + ['A', 'B', 'C']
>>> spam
[1, 2, 3, 'A', 'B', 'C']
```

The * operator can also be used with a list and an integer value to replicate the list.

```
>>> ['X', 'Y', 'Z'] * 3
```

['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']

Q) Briefly explain how values can be removed from Lists with del statement.

The del statement will delete values at an index in a list. All of the values in the list after the deleted value will be moved up one index.

Example:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat']
```

Q) Write a program in python to create a single list to store number of cats that the ser types in and display the names of cats.

```
catNames = []
while True:
        print('Enter the name of cat ' + str(len(catNames) + 1) + ' (Or enter nothing to stop.):')
        name = input()
        if name == '':
                break
        catNames = catNames + [name] # list concatenation
print('The cat names are:')
for name in catNames:
        print(' ' + name)
```

Output:

```
Enter the name of cat 1 (Or enter nothing to stop.):
Zophie
Enter the name of cat 2 (Or enter nothing to stop.):
Pooka
Enter the name of cat 3 (Or enter nothing to stop.):
Simon
Enter the name of cat 4 (Or enter nothing to stop.):
Lady Macbeth
Enter the name of cat 5 (Or enter nothing to stop.):
Fat-tail
Enter the name of cat 6 (Or enter nothing to stop.):
Miss Cleo
Enter the name of cat 7 (Or enter nothing to stop.):
```

The cat names are:
Zophie
Pooka
Simon
Lady Macbeth
Fat-tail
Miss Cleo

Q) Explain the use of for loop with list. Give example.

As we know that, for loops are used to execute a block of code a certain number of times. A for loop repeats the code block once for each value in a list or list-like value.

Example:

```
for i in range(4):
        print(i)
```

Output:

0
1
2
3

Consider the list contains [0, 1, 2, 3]
The following program has the same output as the previous one:

```
for i in [0, 1, 2, 3]:
print(i)
```

A common Python technique is to use range(len(someList)) with a for loop to iterate over the indexes of a list.

Example:

```
>>> supplies = ['pens', 'staplers', 'flame-throwers', 'binders']
>>> for i in range(len(supplies)):
        print('Index ' + str(i) + ' in supplies is: ' + supplies[i])
```

Output:

Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flame-throwers
Index 3 in supplies is: binders

Q) Explain the use of in and not in operators in list with suitable examples.

We can determine whether a value is or isn’ t in a list with the in and not in operators.

Like other operators, in and not in are used in expressions and connect two values:

(i) a value to look for in a list and

(ii) the list where it may be found.

These expressions will evaluate to a Boolean value.

Example:

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
True
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> 'cat' in spam
False
>>> 'howdy' not in spam
False
>>> 'cat' not in spam
True
```

Q) Write a program in python that lets the user type in a pet name and then checks to see whether the name is in a list of pets or not.

```
myPets = ['Zophie', 'Pooka', 'Fat-tail']
print('Enter a pet name:')
name = input()
if name not in myPets:
        print('I do not have a pet named ' + name)
else:
        print(name + ' is my pet.')
```

Output:

```
Enter a pet name:
Footfoot
I do not have a pet named Footfoot
```

Q) Briefly explain the Multiple Assignment in python.

The multiple assignment trick is a shortcut that lets we assign multiple variables with the values in a list in one line of code.

So instead of doing this:

```
>>> cat = ['fat', 'black', 'loud']
>>> size = cat[0]
>>> color = cat[1]
>>> disposition = cat[2]
```

We can type this line of code:

```
>>> cat = ['fat', 'black', 'loud']
>>> size, color, disposition = cat
```

The number of variables and the length of the list must be exactly equal, or Python will give a ValueError:

```
>>> cat = ['fat', 'black', 'loud']
>>> size, color, disposition, name = cat
Traceback (most recent call last):
  File "<pyshell#84>", line 1, in <module>
    size, color, disposition, name = cat
ValueError: need more than 3 values to unpack
```

Q) Explain the Augmented Assignment Operators used in Python and give example.

When assigning a value to a variable, we will frequently use the variable itself. For example, after assigning 42 to the variable spam, we would increase the value in spam by 1 with the following code:

```
>>> spam = 42
>>> spam = spam + 1
>>> spam
43
```

As a shortcut, we can use the augmented assignment operator += to do the same thing:

```
>>> spam = 42
>>> spam += 1
>>> spam
43
```

There are augmented assignment operators for the +, -, *, /, and % operators, described in Table below.

Table. The Augmented Assignment Operators

Augmented assignment statement	Equivalent assignment statement
`spam = spam + 1`	`spam += 1`
`spam = spam - 1`	`spam -= 1`
`spam = spam * 1`	`spam *= 1`
`spam = spam / 1`	`spam /= 1`
`spam = spam % 1`	`spam %= 1`

The += operator can also do string and list concatenation, and the *= operator can do string and list replication.

Example:

```
 >>> spam = 'Hello'
>>> spam += ' world!'
>>> spam
'Hello world!'
>>> bacon = ['Zophie']
>>> bacon *= 3
>>> bacon
['Zophie', 'Zophie', 'Zophie']
```

Q) Define Methods in python. Explain index(), append(), insert(), remove(), len(),and sort() with suitable example.

A method is the same thing as a function, except it is “ called on” a value. For example, if a list value were stored in spam, we will call the index() list method on that list like spam.index('hello'). The method part comes after the value, separated by a period. Each data type has its own set of methods. The list data type, for example, has several useful methods for finding, adding, removing, and otherwise manipulating values in a list.

index()

List values have an index() method that can be passed a value, and if that value exists in the list, the index of the value is returned. If the value isn’ t in the list, then Python produces a ValueError error.

Example:

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> spam.index('hello')
0
```

```
>>> spam.index('heyas')
3
>>> spam.index('howdy howdy howdy')
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    spam.index('howdy howdy howdy')
ValueError: 'howdy howdy howdy' is not in list
```

When there are duplicates of the value in the list, the index of its first appearance is returned.

Enter the following into the interactive shell, and notice that index() returns 1, not 3:

```
>>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
>>> spam.index('Pooka')
1
```

append()

To add new values to a list, use the append() and insert() methods. Enter the following into the interactive shell to call the append() method on a list value stored in the variable spam.

```
 >>> spam = ['cat', 'dog', 'bat']
>>> spam.append('moose')
>>> spam ['cat', 'dog', 'bat', 'moose']
```

The previous append() method call adds the argument to the end of the list.

insert()

The insert() method can insert a value at any index in the list. The first argument to insert() is the index for the new value, and the second argument is the new value to be inserted.

Example:

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.insert(1, 'chicken')
>>> spam
['cat', 'chicken', 'dog', 'bat']
```

remove()

The remove() method is passed the value to be removed from the list it is called on.

Example:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('bat')
```

```
>>> spam
['cat', 'rat', 'elephant']
```

Attempting to delete a value that does not exist in the list will result in a ValueError error.

Example:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('chicken')
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    spam.remove('chicken')
ValueError: list.remove(x): x not in list
```

If the value appears multiple times in the list, only the first instance of the value will be removed.

Example:

```
>>> spam = ['cat', 'bat', 'rat', 'cat', 'hat', 'cat']
>>> spam.remove('cat')
>>> spam
['bat', 'rat', 'cat', 'hat', 'cat']
```

NOTE: The del statement is good to use when we know the index of the value we want to remove from the list. The remove() method is good when we know the value we want to remove from the list.

len()

The len() function will return the number of values that are in a list value passed to it, just like it can count the number of characters in a string value.

Example:

```
>>> spam = ['cat', 'dog', 'moose']
>>> len(spam)
3
```

sort()

Lists of number values or lists of strings can be sorted with the sort() method.

Example:

```
>>> spam = [2, 5, 3.14, 1, -7]
>>> spam.sort()
>>> spam
[-7, 1, 2, 3.14, 5]
>>> spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
```

```
>>> spam.sort()
>>> spam
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

. We can also pass True for the reverse keyword argument to have sort() sort the values in reverse order.

Example:

```
>>> spam.sort(reverse=True)
>>> spam
['elephants', 'dogs', 'cats', 'badgers', 'ants']
```

NOTE: There are three things we should note about the sort() method.

. First, the sort() method sorts the list in place; don’ t try to capture the return value by writing code like spam = spam.sort().

. Second, we cannot sort lists that have both number values and string values in them, since Python doesn’ t know how to compare these values.

Example:

```
>>> spam = [1, 3, 2, 4, 'Alice', 'Bob']
>>> spam.sort()
Traceback (most recent call last):
      File "<pyshell#70>", line 1, in <module>
        spam.sort()
TypeError: unorderable types: str() < int()
```

. Third, sort() uses “ ASCIIbetical order” rather than actual alphabetical order for sorting strings. This means uppercase letters come before lowercase letters. Therefore, the lowercase a is sorted so that it comes after the uppercase Z.

Example:

```
>>> spam = ['Alice', 'ants', 'Bob', 'badgers', 'Carol', 'cats']
>>> spam.sort()
>>> spam ['Alice', 'Bob', 'Carol', 'ants', 'badgers', 'cats']
```

If we need to sort the values in regular alphabetical order, pass str. lower for the key keyword argument in the sort() method call.

Example:

```
>>> spam = ['a', 'z', 'A', 'Z']
>>> spam.sort(key=str.lower)
>>> spam
['a', 'A', 'z', 'Z']
```

This causes the sort() function to treat all the items in the list as if they were lowercase without actually changing the values in the list.

Q) Explain List-like types such as Strings and Tuples with an example.

Lists are data types that represent ordered sequences of values. For example, strings and lists are actually similar, if we consider a string to be a “ list” of single text characters. Many of the things we can do with lists can also be done with strings: indexing; slicing; and using them with for loops, with len(), and with the in and not in operators.

Example:

```
>>> name = 'Zophie'
>>> name[0]
'Z'
>>> name[-2]
'i'
>>> name[0:4]
'Zoph'
>>> 'Zo' in name
True
>>> 'z' in name
False
>>> 'p' not in name
False
>>> for i in name:
        print('* * * ' + i + ' * * *')
```

Output:

```
 * * * Z * * *
* * * o * * *
* * * p * * *
* * * h * * *
* * * i * * *
* * * e * * *
```

Q) Briefly explain Mutable and Immutable data types in python and give example.

A list value is a mutable data type: It can have values added, removed, or changed. However, a string is immutable: It cannot be changed. Trying to reassign a single character in a string results in a TypeError error.

```
>>> name = 'Zophie a cat'
>>> name[7] = 'the'
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
```

```
    name[7] = 'the'
TypeError: 'str' object does not support item assignment
```

The proper way to “ mutate” a string is to use slicing and concatenation to build a new string by copying from parts of the old string.

Example:

```
>>> name = 'Zophie a cat'
>>> newName = name[0:7] + 'the' + name[8:12]
>>> name
'Zophie a cat'
>>> newName
'Zophie the cat'
```

We used [0:7] and [8:12] to refer to the characters that we don’ t wish to replace. Notice that the original 'Zophie a cat' string is not modified because strings are immutable. Although a list value is mutable, the second line in the following code does not modify the list eggs:

Example:

```
>>> eggs = [1, 2, 3]
>>> eggs = [4, 5, 6]
>>> eggs [4, 5, 6]
```

The list value in eggs isn’ t being changed here; rather, an entirely new and different list value ([4, 5, 6]) is overwriting the old list value ([1, 2, 3]).

If we wanted to actually modify the original list in eggs to contain [4, 5, 6], we have to do the following:

```
>>> eggs = [1, 2, 3]
>>> del eggs[2]
>>>del eggs[1]
>>> del eggs[0]
>>> eggs.append(4)
>>> eggs.append(5)
>>> eggs.append(6)
>>> eggs
[4, 5, 6]
```

Changing a value of a mutable data type (like what the del statement and append() method) changes the value in place, since the variable’ s value is not replaced with a new list value.

Q) Briefly explain about Tuple data type and give example.

The tuple data type is almost identical to the list data type, except in two ways. First, tuples are typed with parentheses, (and), instead of square brackets, [and].

Example:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[0] 'hello'
>>> eggs[1:3]
(42, 0.5)
>>> len(eggs)
3
```

But the main way that tuples are different from lists is that tuples, like strings, are immutable. Tuples cannot have their values modified, appended, or removed.

Example:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[1] = 99
Traceback (most recent call last):
   File "<pyshell#5>", line 1, in <module>
      eggs[1] = 99
TypeError: 'tuple' object does not support item assignment
```

If we have only one value in our tuple, we can indicate this by placing a trailing comma after the value inside the parentheses. Otherwise, Python will think we have just typed a value inside regular parentheses. The comma is what lets Python know this is a tuple value.

Enter the following type() function calls into the interactive shell to see the distinction:

```
>>> type(('hello',))
<class ‘ tuple’ >
>>> type(('hello'))
<class ‘ str’ >
```

Q) What are the advantages of tuples.

The advantages of tuples are as follows.

i) We can use tuples to convey to anyone reading our code that we don’ t intend for that sequence of values to change. If we need an ordered sequence of values that never changes, use a tuple.

ii) A second benefit of using tuples instead of lists is that, because they are immutable and their contents don’ t change,

iii) Python can implement some optimizations that make code using tuples slightly faster than code using lists.

Q) Briefly explain that how we can convert types with list() and tuple() functions.

Just like how str(42) will return '42', the string representation of the integer 42, the functions list() and tuple() will return list and tuple versions of the values passed to them.

Example:
Enter the following into the interactive shell, and notice that the return value is of a different data type than the value passed:
>>> **tuple(['cat', 'dog', 5])**
('cat', 'dog', 5)
>>> **list(('cat', 'dog', 5))**
['cat', 'dog', 5]
>>> **list('hello')**
['h', 'e', 'l', 'l', 'o']

Converting a tuple to a list is handy if we need a mutable version of a tuple value.

Q) Briefly explain about references and passing references in python. Give example.
Variables store strings and integer values.
Enter the following into the interactive shell:
>>> spam = 42
>>> cheese = spam
>>> spam = 100
>>> spam
100 >>>
cheese
42
We assign 42 to the spam variable, and then we copy the value in spam and assign it to the variable cheese. When we later change the value in spam to 100, this doesn’ t affect the value in cheese. This is because spam and cheese are different variables that store different values. But lists don’ t work this way. When you assign a list to a variable, we are actually assigning a list reference to the variable. A reference is a value that points to some bit of data, and a list reference is a value that points to a list.
Example:
>>> spam = [0, 1, 2, 3, 4, 5]
>>> cheese = spam
>>> cheese[1] = 'Hello!'
>>> spam
[0, 'Hello!', 2, 3, 4, 5]
>>> cheese
[0, 'Hello!', 2, 3, 4, 5]
Variables will contain references to list values rather than list values themselves. But for strings and integer values, variables simply contain the string or integer value.

Python uses references whenever variables must store values of mutable data types, such as lists or dictionaries. For values of immutable data types such as strings, integers, or tuples, Python variables will store the value itself. Although Python variables technically contain references to list or dictionary values, people often casually say that the variable contains the list or dictionary.

Passing References

References are particularly important for understanding how arguments get passed to functions. When a function is called, the values of the arguments are copied to the parameter variables. For lists, this means a copy of the reference is used for the parameter.

Example:

```
def eggs(someParameter):
        someParameter.append('Hello')

spam = [1, 2, 3]
eggs(spam)
print(spam)
```

Notice that when eggs() is called, a return value is not used to assign a new value to spam. Instead, it modifies the list in place, directly. When run, this program produces the following output:

[1, 2, 3, 'Hello']

Even though spam and someParameter contain separate references, they both refer to the same list. This is why the append('Hello') method call inside the function affects the list even after the function call has returned.

Q) Explain copy() and deepcopy() functions with an example.

Although passing around references is often the handiest way to deal with lists and dictionaries, if the function modifies the list or dictionary that is passed, we may not want these changes in the original list or dictionary value. For this, Python provides a module named copy that provides both the copy() and deepcopy() functions. The first of these, copy.copy(), can be used to make a duplicate copy of a mutable value like a list or dictionary, not just a copy of a reference.

Example:

```
>>> import copy
>>> spam = ['A', 'B', 'C', 'D']
>>> cheese = copy.copy(spam)
>>> cheese[1] = 42
```

```
>>> spam ['A', 'B', 'C', 'D']
>>> cheese ['A', 42, 'C', 'D']
```

Now the spam and cheese variables refer to separate lists, which is why only the list in cheese is modified when we assign 42 at index 7. The reference ID numbers are no longer the same for both variables because the variables refer to independent lists.
If the list you need to copy contains lists, then use the copy.deepcopy() function instead of copy.copy(). The deepcopy() function will copy these inner lists as well.