CHAPTER 1: LISTS

1. The List Data Type
2. Working with Lists
3. Augmented Assignment Operators
4. Methods
5. Example Program: Magic 8 Ball with a List
6. List-like Types: Strings and Tuples
7. References

1.1 The List Data Type

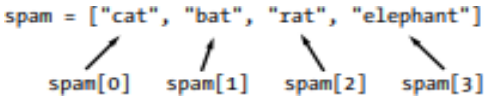
- A list is a value that contains multiple values in an ordered sequence.
- A list value looks like this: ['cat', 'bat', 'rat', 'elephant'].
- A list begins with an opening square bracket and ends with a closing square bracket, [].
- Values inside the list are also called items and are separated with commas.

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

- The spam variable ❶ is still assigned only one value: the list value(contains multiple values).
- The value [] is an empty list that contains no values, similar to '', the empty string.

Getting Individual Values in a List with Indexes

- Say you have 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 first value in the list is at index 0, the second value is at index 1, and the third value is at index 2, and so on.
- For example, type the following expressions into the interactive shell.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0]
'cat'
>>> spam[1]
'bat'
>>> spam[2]
'rat'
>>> spam[3]
'elephant'
>>> ['cat', 'bat', 'rat', 'elephant'][3]
'elephant'
❶ >>> 'Hello ' + spam[0]
❷ 'Hello cat'
>>> 'The ' + spam[1] + ' ate the ' + spam[0] + '.'
'The bat ate the cat.'
```

- The expression 'Hello ' + spam[0] evaluates to 'Hello ' + 'cat' because spam[0] evaluates to the string 'cat'. This expression in turn evaluates to the string value 'Hello cat'.
- If we use an index that exceeds the number of values in the list value then, python gives IndexError.

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[10000]
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    spam[10000]
IndexError: list index out of range
```

- Indexes can be only integer values, not floats. The following example will cause a TypeError error:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1]
'bat'
>>> spam[1.0]
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    spam[1.0]
TypeError: list indices must be integers, not float
>>> spam[int(1.0)]
'bat'
```

- Lists can also contain other list values. The values in these lists of lists can be accessed using multiple indexes.

```
>>> spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
>>> spam[0]
['cat', 'bat']
>>> spam[0][1]
'bat'
>>> spam[1][4]
50
```

- The first index dictates which list value to use, and the second indicates the value within the list value. **Ex**, spam[0][1] prints 'bat', the second value in the first list.

Negative Indexes

- We 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.

```
>>> 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.'
```

Getting Sublists with Slices

- An index will 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.
- **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 (but will not include the value at the second index).

```
>>> 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.

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

Getting a List's Length with len()

- The len() function will return the number of values that are in a list value.

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

Changing Values in a List with Indexes

- We can also use an index of a list to change the value at that index. **Ex:** spam[1] = 'aardvark' means "Assign the value at index 1 in the list spam to the string 'aardvark'."

```
>>> 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]
```

List Concatenation and List Replication

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

```
>>> [1, 2, 3] + ['A', 'B', 'C']
[1, 2, 3, 'A', 'B', 'C']
>>> ['X', 'Y', 'Z'] * 3
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
>>> spam = [1, 2, 3]
>>> spam = spam + ['A', 'B', 'C']
>>> spam
[1, 2, 3, 'A', 'B', 'C']
```

Removing Values from Lists with del Statements

- The del statement will delete values at an index in a list.

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

- The del statement can also be used to delete a variable After deleting if we try to use the variable, we will get a NameError error because the variable no longer exists.
- In practice, you almost never need to delete simple variables.
- The del statement is mostly used to delete values from lists.

1.2 Working with Lists

- When we first begin writing programs, it's tempting to create many individual variables to store a group of similar values.

```
catName1 = 'Zophie'
catName2 = 'Pooka'
catName3 = 'Simon'
catName4 = 'Lady Macbeth'
catName5 = 'Fat-tail'
catName6 = 'Miss Cleo'
```

- Which is bad way to write code because it leads to have a duplicate code in the program.

```
print('Enter the name of cat 1:')
catName1 = input()
print('Enter the name of cat 2:')
catName2 = input()
print('Enter the name of cat 3:')
catName3 = input()
print('Enter the name of cat 4:')
catName4 = input()
print('Enter the name of cat 5:')
catName5 = input()
print('Enter the name of cat 6:')
catName6 = input()
print('The cat names are:')
print(catName1 + ' ' + catName2 + ' ' + catName3 + ' ' + catName4 + ' ' +
catName5 + ' ' + catName6)
```

- Instead of using multiple, repetitive variables, we can use a single variable that contains a list value.
- **For Ex:** The following program uses a single list and it can store any number of cats that the user types in.
- Program:

```
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
```

- The benefit of using a list is that our data is now in a structure, so our program is much more flexible in processing the data than it would be with several repetitive variables.

Using for Loops with Lists

- A for loop repeats the code block once for each value in a list or list-like value.

Program

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

Output:

```
0
1
2
3
```

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

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

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
```

- The code in the loop will access the index (as the variable i), the value at that index (as supplies[i]) and range(len(supplies)) will iterate through all the indexes of supplies, no matter how many items it contains.

The in and not in Operators

- We can determine whether a value is or isn't in a list with the in and not in operators.
- **in** and **not in** are used in expressions and connect two values: a value to look for in a list and the list where it may be found and these expressions will evaluate to a Boolean value.

```
>>> '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
```

- The following program lets the user type in a pet name and then checks to see whether the name is in a list of pets.

Program

```
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
```

The Multiple Assignment Trick

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

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

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

- Instead of left-side program we could type the right-side program to assignment multiple variables but the number of variables and the length of the list must be exactly equal, or Python will give you 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
```

1.3 Augmented Assignment Operators

- When assigning a value to a variable, we will frequently use the variable itself.

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

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

- Instead of left-side program we could use right-side program i.e., with the augmented assignment operator += to do the same thing as a shortcut.
- The Augmented Assignment Operators are listed in the below table:

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.

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

1.4 Methods

- A method is same as a function, except it is "called on" a value.
- The method part comes after the value, separated by a period.
- Each data type has its own set of methods.
- The list data type has several useful methods for finding, adding, removing, and manipulating values in a list.

Finding a Value in a List with the index() Method

- 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.

```
>>> 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.

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

Adding Values to Lists with the append() and insert() Methods

- To add new values to a list, use the append() and insert() methods.
- The append() method call adds the argument to the end of the list.

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

- 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.

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

- Methods belong to a single data type.
- The append() and insert() methods are list methods and can be called only on list values, not on other values such as strings or integers.

```
>>> eggs = 'hello'
>>> eggs.append('world')
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    eggs.append('world')
AttributeError: 'str' object has no attribute 'append'
>>> bacon = 42
>>> bacon.insert(1, 'world')
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    bacon.insert(1, 'world')
AttributeError: 'int' object has no attribute 'insert'
```

Removing Values from Lists with remove()

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

```
>>> 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.

```
>>> 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.

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

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

Sorting the Values in a List with the sort() Method

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

```
>>> 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']
```

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

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

- There are three things you should note about the sort() method.
  - **First,** the sort() method sorts the list in place; don't try to return value by writing code like spam = spam.sort().
  - **Second,** we cannot sort lists that have both number values and string values in them.

```
>>> 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(upper case)" rather than actual alphabetical order(lower case) for sorting strings.

```
>>> 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.

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

1.5 Example Program: Magic 8 Ball with a List

- We can write a much more elegant version of the Magic 8 Ball program. Instead of several lines of nearly identical elif statements, we can create a single list.

```
import random

messages = ['It is certain',
    'It is decidedly so',
    'Yes definitely',
    'Reply hazy try again',
    'Ask again later',
    'Concentrate and ask again',
    'My reply is no',
    'Outlook not so good',
    'Very doubtful']

print(messages[random.randint(0, len(messages) - 1)])
```

- The expression you use as the index into messages: random .randint(0, len(messages) - 1). This produces a random number to use for the index, regardless of the size of messages. That is, you'll get a random number between 0 and the value of len(messages) - 1.

Exceptions to Indentation Rules in Python

- The amount of indentation for a line of code tells Python what block it is in.
- lists can actually span several lines in the source code file. The indentation of these lines do not matter; Python knows that until it sees the ending square bracket, the list is not finished.

```
spam = ['apples',
    'oranges',
                    'bananas',
'cats']
print(spam)
```

- We can also split up a single instruction across multiple lines using the \ line continuation character at the end.

```
print('Four score and seven ' + \
      'years ago...')
```

1.6 List-like Types: Strings and Tuples

- Lists aren't the only data types that represent ordered sequences of values.
- **Ex,** we can also do these with strings: indexing; slicing; and using them with for loops, with len(), and with the in and not in operators.

```
>>> 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 + ' * * *')

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

Mutable and Immutable Data Types

String

- 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.

```
>>> 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.

List

- A list value is a mutable data type: It can have values added, removed, or changed.

```
>>> 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]).

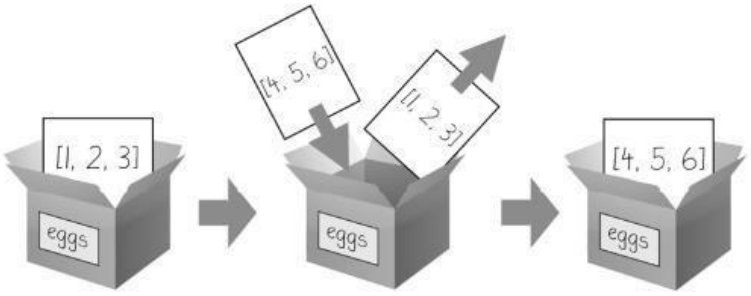


Figure: When eggs = [4, 5, 6] is executed, the contents of eggs are replaced with a new list value.

- If we want to modify the original list in eggs to contain [4, 5, 6], you would have to delete the items in that and then add items to it.

```
>>> 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]
```

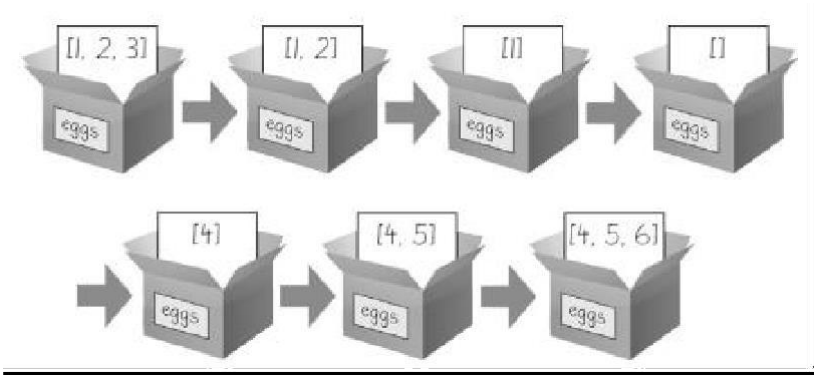


Figure: The del statement and the append() method modify the same list value in place.

The Tuple Data Type

- 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].

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

- **Second**, benefit of using tuples instead of lists is that, because they are immutable and their contents don't change. Tuples cannot have their values modified, appended, or removed.

```
>>> 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 you have only one value in your tuple, you can indicate this by placing a trailing comma after the value inside the parentheses.

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

Converting Types with the list() and tuple() Functions

- The functions list() and tuple() will return list and tuple versions of the values passed to them.

```
>>> 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 you need a mutable version of a tuple value.

1.7 References

- As we've seen, variables store strings and integer values.

```
>>> 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 works differently. When we 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.

```
❶ >>> spam = [0, 1, 2, 3, 4, 5]
❷ >>> cheese = spam
❸ >>> cheese[1] = 'Hello!'
  >>> spam
  [0, 'Hello!', 2, 3, 4, 5]
  >>> cheese
  [0, 'Hello!', 2, 3, 4, 5]
```

- When we create the list ❶, we assign a reference to it in the spam variable. But the next line copies only the list reference in spam to cheese, not the list value itself. This means the values stored in spam and cheese now both refer to the same list.
- There is only one underlying list because the list itself was never actually copied. So when we modify the first element of cheese, we are modifying the same list that spam refers to.
- List variables don't actually contain lists—they contain references to lists.

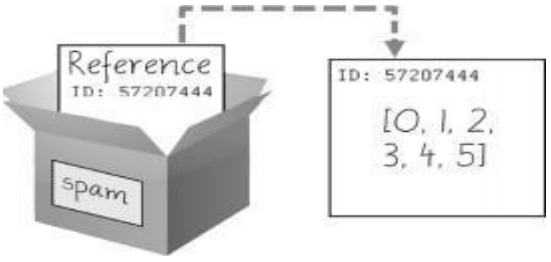


Figure: spam = [0, 1, 2, 3, 4, 5] stores a reference to a list, not the actual list.

- The reference in spam is copied to cheese. Only a new reference was created and stored in cheese, not a new list.

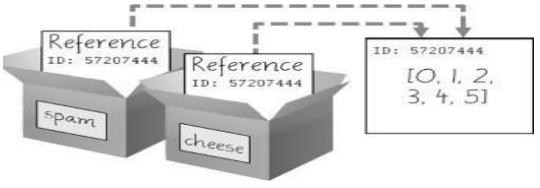


Figure: spam = cheese copies the reference, not the list

- When we alter the list that cheese refers to, the list that spam refers to is also changed, because both cheese and spam refer to the same list.

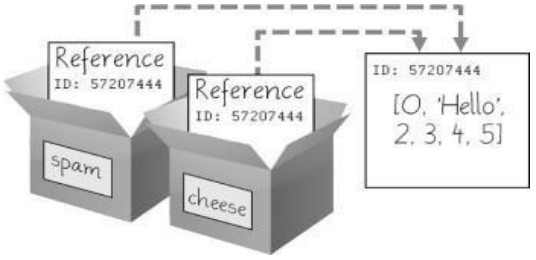


Figure: cheese[1] = 'Hello!' modifies the list that both variables refer to

- Variables will contain references to list values rather than list values themselves.
- But for strings and integer values, variables will 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.

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.

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

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

Program

```
[1, 2, 3, 'Hello']
```

Output

- when eggs() is called, a return value is not used to assign a new value to spam.
- 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.

The copy Module's copy() and deepcopy() Functions

- 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.
- **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.
- Now the spam and cheese variables refer to separate lists, which is why only the list in cheese is modified when you assign 42 at index 1.
- The reference ID numbers are no longer the same for both variables because the variables refer to independent lists.

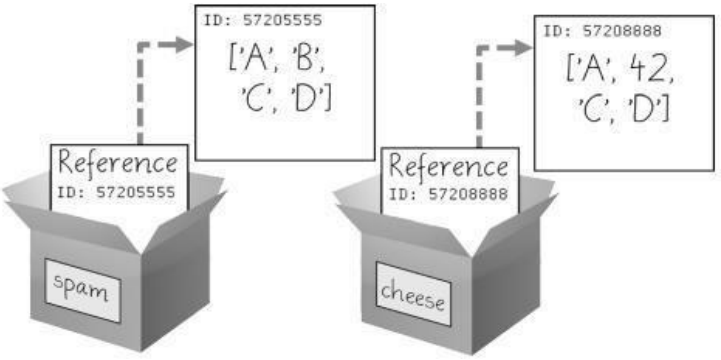


Figure: cheese = copy.copy(spam) creates a second list that can be modified independently of the first.

- 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.

CHAPTER 2: DICTIONARIES AND STRUCTURING DATA

1. The Dictionary Data Type
2. Pretty Printing
3. Using Data Structures to Model Real-World Things.

2.1 The Dictionary Data Type

- A dictionary is a collection of many values. Indexes for dictionaries can use many different data types, not just integers. Indexes for dictionaries are called keys, and a key with its associated value is called a key-value pair.
- A dictionary is typed with braces, {}.

```
>>> myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
```

- This assigns a dictionary to the myCat variable. This dictionary's keys are 'size', 'color', and 'disposition'. The values for these keys are 'fat', 'gray', and 'loud', respectively. You can access these values through their keys:

```
>>> myCat['size']
'fat'
>>> 'My cat has ' + myCat['color'] + ' fur.'
'My cat has gray fur.'
```

- Dictionaries can still use integer values as keys, but they do not have to start at 0 and can be any number.

```
>>> spam = {12345: 'Luggage Combination', 42: 'The Answer'}
```

Dictionaries vs. Lists

- Unlike lists, items in dictionaries are unordered.
- The first item in a list named spam would be spam[0]. But there is no "first" item in a dictionary. While the order of items matters for determining whether two lists are the same, it does not matter in what order the key-value pairs are typed in a dictionary.

```
>>> spam = ['cats', 'dogs', 'moose']
>>> bacon = ['dogs', 'moose', 'cats']
>>> spam == bacon
False
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> eggs == ham
True
```

- Trying to access a key that does not exist in a dictionary will result in a KeyError error message, much like a list's "out-of-range" IndexError error message.

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> spam['color']
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    spam['color']
KeyError: 'color'
```

- We can have arbitrary values for the keys that allows us to organize our data in powerful ways.
- **Ex:** we want to store data about our friends' birthdays. We can use a dictionary with the names as keys and the birthdays as values.

```
❶ birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}

  while True:
      print('Enter a name: (blank to quit)')
      name = input()
      if name == '':
          break

❷     if name in birthdays:
❸         print(birthdays[name] + ' is the birthday of ' + name)
      else:
          print('I do not have birthday information for ' + name)
          print('What is their birthday?')
          bday = input()
❹         birthdays[name] = bday
          print('Birthday database updated.')
```

Program

```
Enter a name: (blank to quit)
Alice
Apr 1 is the birthday of Alice
Enter a name: (blank to quit)
Eve
I do not have birthday information for Eve
What is their birthday?
Dec 5
Birthday database updated.
Enter a name: (blank to quit)
Eve
Dec 5 is the birthday of Eve
Enter a name: (blank to quit)
```

Output

- We create an initial dictionary and store it in birthdays **1**.
- We can see if the entered name exists as a key in the dictionary with the in keyword **2**.
- If the name is in the dictionary, we access the associated value using square brackets **3**; if not, we can add it using the same square bracket syntax combined with the assignment operator **4**.

The keys(), values(), and items() Methods

- There are three dictionary methods that will return list-like values of the dictionary's keys, values, or both keys and values: keys(), values(), and items().
- Data types (dict_keys, dict_values, and dict_items, respectively) can be used in for loops.

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
        print(v)

red
42
```

- A for loop can iterate over the keys, values, or key-value pairs in a dictionary by using keys(), values(), and items() methods.
- The values in the dict_items value returned by the items() method are tuples of the key and value.

```
>>> for k in spam.keys():
        print(k)

color
age
>>> for i in spam.items():
        print(i)

('color', 'red')
('age', 42)
```

- If we want a true list from one of these methods, pass its list-like return value to the list() function.

```
>>> spam = {'color': 'red', 'age': 42}
>>> spam.keys()
dict_keys(['color', 'age'])
>>> list(spam.keys())
['color', 'age']
```

- The list(spam.keys()) line takes the dict_keys value returned from keys() and passes it to list(), which then returns a list value of ['color', 'age'].
- We can also use the multiple assignment trick in a for loop to assign the key and value to separate variables.

```
>>> spam = {'color': 'red', 'age': 42}
>>> for k, v in spam.items():
       print('Key: ' + k + ' Value: ' + str(v))

Key: age Value: 42
Key: color Value: red
```

Checking Whether a Key or Value Exists in a Dictionary

- We can use the **in** and **not in** operators to see whether a certain key or value exists in a dictionary.

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> 'name' in spam.keys()
True
>>> 'Zophie' in spam.values()
True
>>> 'color' in spam.keys()
False
>>> 'color' not in spam.keys()
True
>>> 'color' in spam
False
```

The get() Method

- Dictionaries have a get() method that takes two arguments:
  - The key of the value to retrieve and
  - A fallback value to return if that key does not exist.

```
>>> picnicItems = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'
'I am bringing 2 cups.'
>>> 'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'
'I am bringing 0 eggs.'
```

The setdefault() Method

- To set a value in a dictionary for a certain key only if that key does not already have a value.

```
spam = {'name': 'Pooka', 'age': 5}
if 'color' not in spam:
    spam['color'] = 'black'
```

- The setdefault() method offers a way to do this in one line of code.
- Setdeafault() takes 2 arguments:
  - The first argument is the key to check for, and
  - The second argument is the value to set at that key if the key does not exist. If the key does exist, the setdefault() method returns the key's value.

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
>>> spam.setdefault('color', 'white')
'black'
>>> spam
{'color': 'black', 'age': 5, 'name': 'Pooka'}
```

- The first time setdefault() is called, the dictionary in spam changes to {'color': 'black', 'age': 5, 'name': 'Pooka'}. The method returns the value 'black' because this is now the value set for the key 'color'. When spam.setdefault('color', 'white') is called next, the value for that key is not changed to 'white' because spam already has a key named 'color'.

- **Ex:** program that counts the number of occurrences of each letter in a string.

```
message = 'It was a bright cold day in April, and the clocks were striking thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

print(count)
```

- The program loops over each character in the message variable's string, counting how often each character appears.
- The setdefault() method call ensures that the key is in the count dictionary (with a default value of 0), so the program doesn't throw a KeyError error when count[character] = count[character] + 1 is executed.

Output:

```
{' ': 13, ',': 1, '.': 1, 'A': 1, 'I': 1, 'a': 4, 'c': 3, 'b': 1, 'e': 5, 'd': 3, 'g': 2, 'i':
6, 'h': 3, 'k': 2, 'l': 3, 'o': 2, 'n': 4, 'p': 1, 's': 3, 'r': 5, 't': 6, 'w': 2, 'y': 1}
```

2.2 Pretty Printing

- Importing pprint module will provide access to the pprint() and pformat() functions that will "pretty print" a dictionary's values.
- This is helpful when we want a cleaner display of the items in a dictionary than what print() provides and also it is helpful when the dictionary itself contains nested lists or dictionaries..

Program: counts the number of occurrences of each letter in a string.

```
import pprint
message = 'It was a bright cold day in April, and the clocks were striking
thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0)
    count[character] = count[character] + 1

pprint.pprint(count)
```

Output:

```
{'  ': 13,
 ',': 1,
 '.': 1,
 'A': 1,
 'I': 1,
 'a': 4,
 'b': 1,
 'c': 3,
 'd': 3,
 'e': 5,
 'g': 2,
 'h': 3,
 'i': 6,
 'k': 2,
 'l': 3,
 'n': 4,
 'o': 2,
 'p': 1,
 'r': 5,
 's': 3,
 't': 6,
 'w': 2,
 'y': 1}
```

- If we want to obtain the prettified text as a string value instead of displaying it on the screen, call pprint.pformat().

```
pprint.pprint(someDictionaryValue)
print(pprint.pformat(someDictionaryValue))
```

2.3 Using Data Structures to Model Real-World Things

A Tic-Tac-Toe Board

- A tic-tac-toe board looks like a large hash symbol (#) with nine slots that can each contain an X, an O, or a blank. To represent the board with a dictionary, we can assign each slot a string-value key as shown in below figure.

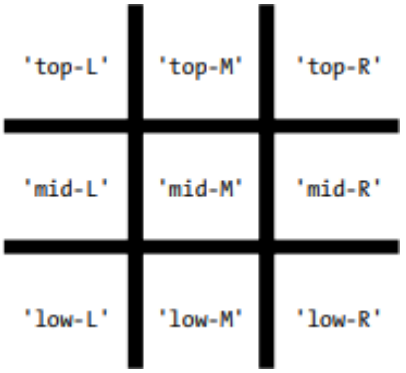


Figure: The slots of a tic-tactoe board with their corresponding keys

- We can use string values to represent what's in each slot on the board: 'X', 'O', or ' ' (a space character).
- To store nine strings. We can use a dictionary of values for this.
  - The string value with the key 'top-R' can represent the top-right corner,
  - The string value with the key 'low-L' can represent the bottom-left corner,

- The string value with the key 'mid-M' can represent the middle, and so on.

- Store this board-as-a-dictionary in a variable named theBoard.

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

- The data structure stored in the theBoard variable represents the tic-tactoe board in the below Figure.

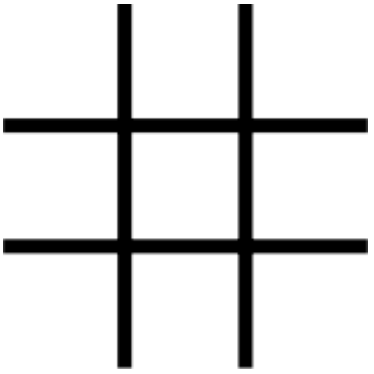


Figure: An empty tic-tac-toe board

- Since the value for every key in theBoard is a single-space string, this dictionary represents a completely clear board. If player X went first and chose the middle space, you could represent that board with this dictionary as shown below:

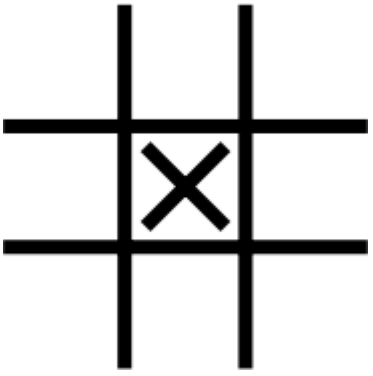


Figure: A first move

- A board where player O has won by placing Os across the top might look like this:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O',
            'mid-L': 'X', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}
```

- The data structure in theBoard now represents the tic-tac-toe board in the below Figure.

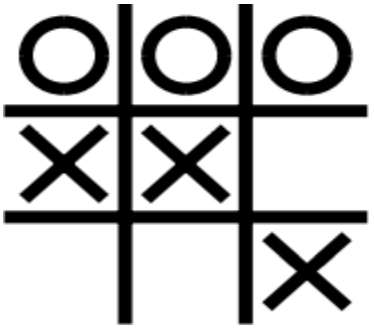


Figure: Player O wins.

- The player sees only what is printed to the screen, not the contents of variables.
- The tic-tac-toe program is updated as below.

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
printBoard(theBoard)
```

Output:

- The printBoard() function can handle any tic-tac-toe data structure you pass it.

Program

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O', 'mid-L': 'X', 'mid-M':
'X', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}

def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
printBoard(theBoard)
```

Output:

```
O|O|O
-+-+-
X|X|
-+-+-
 |  |X
```

- Now we created a data structure to represent a tic-tac-toe board and wrote code in printBoard() to interpret that data structure, we now have a program that "models" the tic-tac-toe board.

Program: allows the players to enter their moves.

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ', 'mid-L': ' ', 'mid-M': '
', 'mid-R': ' ', 'low-L': ' ', 'low-M': ' ', 'low-R': ' '}

def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])
turn = 'X'
for i in range(9):
❶   printBoard(theBoard)
    print('Turn for ' + turn + '. Move on which space?')
❷   move = input()
❸   theBoard[move] = turn
❹   if turn == 'X':
        turn = 'O'
    else:
        turn = 'X'
printBoard(theBoard)
```

Output:

```
  |  |
-+-+-
  |  |
-+-+-
  |  |
Turn for X. Move on which space?
mid-M
  |  |
-+-+-
  |X|
-+-+-
  |  |
Turn for O. Move on which space?
low-L
  |  |
-+-+-
  |X|
-+-+-
O|  |

--snip--

O|O|X
-+-+-
X|X|O
-+-+-
O|  |X
Turn for X. Move on which space?
low-M
O|O|X
-+-+-
X|X|O
-+-+-
O|X|X
```

- The new code prints out the board at the start of each new turn **1**, gets the active player's move **2**, updates the game board accordingly **3**, and then swaps the active player **4** before moving on to the next turn.

Nested Dictionaries and Lists

- We can have program that contains dictionaries and lists which in turn contain other dictionaries and lists.
- Lists are useful to contain an ordered series of values, and dictionaries are useful for associating keys with values.

Program: which contains nested dictionaries in order to see who is bringing what to a picnic.

```
allGuests = {'Alice': {'apples': 5, 'pretzels': 12},
             'Bob': {'ham sandwiches': 3, 'apples': 2},
             'Carol': {'cups': 3, 'apple pies': 1}}

def totalBrought(guests, item):
    numBrought = 0
❶   for k, v in guests.items():
❷       numBrought = numBrought + v.get(item, 0)
    return numBrought

print('Number of things being brought:')
print(' - Apples         ' + str(totalBrought(allGuests, 'apples')))
print(' - Cups           ' + str(totalBrought(allGuests, 'cups')))
print(' - Cakes          ' + str(totalBrought(allGuests, 'cakes')))
print(' - Ham Sandwiches ' + str(totalBrought(allGuests, 'ham sandwiches')))
print(' - Apple Pies     ' + str(totalBrought(allGuests, 'apple pies')))
```

- Inside the totalBrought() function, the for loop iterates over the keyvalue pairs in guests **1.**
- Inside the loop, the string of the guest's name is assigned to k, and the dictionary of picnic items they're bringing is assigned to v.
- If the item parameter exists as a key in this dictionary, it's value (the quantity) is added to numBrought **2**.

- If it does not exist as a key, the get() method returns 0 to be added to numBrought.

Output:

```
Number of things being brought:
- Apples 7
- Cups 3
- Cakes 0
- Ham Sandwiches 3
- Apple Pies     1
```