

Module-2

Lists, Dictionaries and Structuring Data

LISTS

- A list is a collection which is ordered and changeable.
- Allows duplicate members.
- In Python lists are written with square brackets.

Example: Create a List

```
>>> fruitlist = ["apple", "banana", "cherry"]
```

A tuple is a collection which is ordered and unchangeable.

- In Python tuples are written with round brackets.

Example: create a tuple

```
>>> fruittuple = ("apple", "banana", "cherry")
```

- *Lists and tuples* can contain *multiple values* which makes it easier to write programs that handle large amounts of data.
- Since ***lists themselves can contain other lists***, we can use them to arrange data into ***hierarchical structure***.

Example: for list value ['apple', 'banana', 'cherry']

1. It is a string values & typed in pair of single quote characters where the string begins & ends.
2. A list begins with an opening square brackets and ends with closed square brackets [].
3. The values inside the list are also called items [1, 2, 3, 4, 5] elements are separated by commas.

Introduction to Python Programming

Example-1:

```
>>> number=[1,2,3,4,5]
>>> number
[1, 2, 3, 4, 5]
```

Example-2:

```
>>> list=['Hello',3.1412,True, None,42]
>>> list
['Hello', 3.1412, True, None, 42]
```

Example-3


```
>>> fruitlist = ["apple", "banana", "cherry"]
>>> fruitlist
['apple', 'banana', 'cherry']
```

From the above **example-3**, the **fruitlist** variable is still *assigned only one value: the list value*. But the **list** value itself contains **other values**. The value **[]** is an **empty list** that contains no values, similar to, the **empty string**.

Getting Individual Values in a List with Indexes

Example:

```
fruitlist = ["apple", "banana", "cherry"]
```

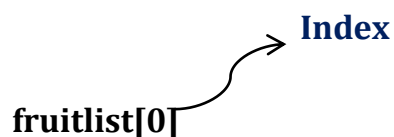


```
fruitlist[0]  fruitlist[1]  fruitlist[2]
```

Fig 1: A list value stored in the variable **fruitlist**, showing which value each index refers to.

- The above list **["apple", "banana", "cherry"]** stored in a variable named **fruitlist**.
- The python code **fruitlist[0]** would evaluate to apple, **fruitlist[1]** would evaluate to banana and **fruitlist[2]** would evaluate to cherry. The **integer** inside the square bracket that follows the list is called **index**.

Example:



```
fruitlist[0]
```

Index

Introduction to Python Programming

- In the above example, 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-1 shows** a list value assigned to `fruitlist`, along with what the index expressions would evaluate to.

For example,

```
>>> fruitlist = ["apple", "banana", "cherry"]
>>> fruitlist[0]
'apple'
>>> fruitlist[1]
'banana'
>>> fruitlist[2]
'cherry'
```

```
>>> ["apple", "banana", "cherry"][2]
'cherry'
>>> "hello " + fruitlist[0]
'hello apple'
>>> "hello " + fruitlist[1]
'hello banana'
>>> "hello " + fruitlist[2]
'hello cherry'
```

```
>>> 'The ' + fruitlist[0] + ' and ' + fruitlist[1] + ' are the good fruits' '.'
'The apple and banana are the good fruits.'
```

- The expression **'hello ' + fruitlist[0]** evaluates to **'hello ' + 'apple'** because **fruitlist[0]** evaluates to the string **'apple'**. This expression in turn evaluates to the string value **'hello apple'**.
- Similarly the expression **'hello ' + fruitlist[1]** evaluates to **'hello ' + 'banana'** because **fruitlist[1]** evaluates to the string **'banana'**. This expression in turn evaluates to the string value **'hello banana'**.

Introduction to Python Programming

- Similarly the expression `'hello ' + fruitlist[2]` evaluates to `'hello ' + 'cherry'` because `fruitlist[2]` evaluates to the string `'cherry'`. This expression in turn evaluates to the string value `'hello cherry '`.

For Example

Python will give you an [IndexError](#) error message if you **use an index that exceeds the number of values in your list value**.

```
>>> fruitlist = ["apple", "banana", "cherry"]
>>> fruitlist
['apple', 'banana', 'cherry']
>>> fruitlist[10]
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    fruitlist[10]
IndexError: list index out of range
```

For Example

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

```
>>> fruitlist = ["apple", "banana", "cherry"]
>>> fruitlist
['apple', 'banana', 'cherry']
>>> fruitlist[1]
'banana'
>>> fruitlist[1.0]
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    fruitlist[1.0]
TypeError: list indices must be integers or slices, not float
>>> fruitlist[int(1.0)]
'banana'
```

Introduction to Python Programming

For Example

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

```
>>> fruitlist = [ ["apple", "banana", "cherry"], [10,20,30,40]]
>>> fruitlist[0]
['apple', 'banana', 'cherry']
>>> fruitlist[0][1]
'banana'
>>> fruitlist[1][2]
30
>>> fruitlist[1][2]
30
>>> fruitlist[1][3]
40
```

The first index dictates *which list value to use*, and the second indicates the *value within the list value*. For example, `fruitlist[0][1]` prints **'banana'**, the second value in the first list. *If you only use one index*, the program will print the full list value at that index.

Example:

```
>>> fruitlist=[ ['banana', 'apple'], [10,20,30]]
>>> fruitlist[0]
['banana', 'apple']
>>> fruitlist[1]
[10, 20, 30]
>>> fruitlist[0][1]
'apple'
>>> fruitlist[0][0]
'banana'
>>> fruitlist[1][0]
10
>>> fruitlist[1][1]
20
>>> fruitlist[1][2]
30
```

Negative Indexing

Negative indexing means ***beginning from the end***, **-1** refers to the ***last item***, **-2** refers to the ***second last item*** and **-3** refers to the ***first item***.

Example:

```
>>> fruitlist = ["apple", "banana", "cherry"]
>>> fruitlist[-1]
'cherry'
>>> fruitlist[-2]
'banana'
>>> fruitlist[-3]
'apple'
```

Getting Sublists with Slices

- An **index** can get a **single value** from a **list**, where as
- 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***.
- Notice the **difference between indexes and slices**.
 1. **fruitlist[1]** is a list with an index (**one integer**).
 2. **fruitlist[1:2]** is a list with a slice (**two integers**).

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***.

Finally a **slice evaluates to a new list value**.

Example:

```
>>> fruitlist = ["apple", "banana", "cherry"]
>>> fruitlist[1:2]
['banana']
>>> fruitlist[0:-1]
['apple', 'banana']
```

Introduction to Python Programming

As a *shortcut*, you 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*.

```
>>> fruitlist = ["apple", "banana", "cherry"]
>>> fruitlist[1:]
['banana', 'cherry']
>>> fruitlist[:2]
['apple', 'banana']
>>> fruitlist[:]
['apple', 'banana', 'cherry']
```

Getting a List's Length with 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.

```
>>> fruitlist = ["apple", "banana", "cherry"]
>>> len(fruitlist)
3
>>> fruitlist = ["apple", "banana"]
>>> len(fruitlist)
2
```

Changing Values in a List with Indexes

Normally a variable name goes on the left side of an assignment statement, (like **fruitlist=orange**). However, **you can also use *an index of a list* to *change the value at that index***.

For example,

```
>>> fruitlist = ["apple", "banana", "cherry"]
>>> fruitlist
['apple', 'banana', 'cherry']
```

Introduction to Python Programming

For example

`fruitlist[2] = 'orange'` , means “Assign the value at index 2 in the list `fruitlist` to the string 'orange'.” (Like `fruitlist[2]='orange'`) similarly `fruitlist[1]='sapota'` , means assign the value at index 1 in the list `fruitlist` to the string 'sapota' (Like `fruitlist[1]='sapota'`)

```
>>> fruitlist = ["apple", "banana", "cherry"]
>>> fruitlist
['apple', 'banana', 'cherry']
>>> fruitlist[2]= 'orange'
>>> fruitlist
['apple', 'banana', 'orange']
>>> fruitlist[1]='sapota'
>>> fruitlist
['apple', 'sapota', 'orange']
```

- In the below *fruitlist*, the fruit name is changed using the index value. Like the index value 2 is orange, and the changing the orange fruit to sapota fruit.

Example:

```
>>> fruitlist = ["apple", "sapota", "orange"]
>>> fruitlist
['apple', 'sapota', 'orange']
>>> fruitlist[2]=fruitlist[1]
>>> fruitlist
['apple', 'sapota', 'sapota']
```

Example: `fruitlist = ["apple", "sapota", "sapota"]`

- Similarly, changing the fruit name using the index value. Like the index value 2 is sapota, and the changing the sapota fruit to orange using the *negative index value*.

```
>>> fruitlist = ["apple", "sapota", "sapota"]
>>> fruitlist
['apple', 'sapota', 'sapota']
>>> fruitlist[-1]='orange'
>>> fruitlist
['apple', 'sapota', 'orange']
```

Introduction to Python Programming

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*.

Example:

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

>>> num = [1, 2, 3]
>>> num = num + [1, 2, 3]
>>> num
[1, 2, 3, 1, 2, 3]
```

Removing Values from Lists with del Statements

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:

```
>>> fruitlist = ["apple", "sapota", "orange"]
>>> fruitlist
['apple', 'sapota', 'orange']
>>> del fruitlist[2]
>>> fruitlist
['apple', 'sapota']
>>> del fruitlist[1]
>>> fruitlist
['apple']
```

- The **del** statement can also be **used on a simple variable to delete it**, as if it were an ***“unassignment”*** statement. If you try to use the variable after deleting it, you will get a ***NameError*** error because, the variable no longer exists.
- The **del** statement is mostly used to delete values from lists.

WORKING WITH LISTS

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

For example, if I wanted to store the names of fruits.

Fruit-1='apple'

Fruit-2='banana'

Fruit-3='orange'

Fruit-3='mango'

Fruit-3='cherry'

It turns out that this is a **bad way to write code**. For one thing, if the number of fruits changes, your program will never be able to store more fruits than you have variables. These types of programs also have a lot of duplicate or nearly identical code in them. Consider how much duplicate code is in the following program, which you should enter into the file editor.

```
print('Enter the name of fruit 1:')
fruitName1 = input()
print('Enter the name of fruit 2:')
fruitName2 = input()
print('Enter the name of fruit 3:')
fruitName3 = input()
print('Enter the name of fruit 4:')
fruitName4 = input()
print('Enter the name of fruit 5:')
fruitName5 = input()

#display the fruit names
print('The fruit names are:')
print(fruitName1 + ',' + fruitName2 + ',' + fruitName3 + ',' + fruitName4
      + ',' + fruitName5 )
```

Introduction to Python Programming

- Instead of using **multiple, repetitive variables**, you can use a **single variable** that contains a list value.

For example, here's a **new and improved version of the *above* program**. This new version uses a **single list** and can store any number of fruits that the user types in.

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

The output of this program is:

```
Enter the name of fruit 1 (Or enter nothing to stop.):
apple
Enter the name of fruit 2 (Or enter nothing to stop.):
banana
Enter the name of fruit 3 (Or enter nothing to stop.):
orange
Enter the name of fruit 4 (Or enter nothing to stop.):
mango
Enter the name of fruit 5 (Or enter nothing to stop.):
Cherry
```

The fruit names are:

```
Apple
Banana
Orange
Mango
Cherry
```

Introduction to Python Programming

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

Using for Loops with Lists

In the previous module, you learned about using for loops to execute a block of code a certain number of times. Technically, a **for** loop repeats the code block once for each value in a list or list-like value.

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

The output of the code is:

```
0  
1  
2  
3
```

- This is because the return value from **range(4)** is a **list-like value** that Python considers similar to **[0, 1, 2, 3]**.
- The following program has the **same output** as the previous one:

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

- What the previous **for** loop actually does is loop through its clause with the variable **i** set to a successive value in the **[0, 1, 2, 3]** list in each iteration.
- A common Python technique is to use **range(len(someList))** with a **for** loop to *iterate over the indexes of a list*.

For 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

- Using **range(len(supplies))** in the previously shown **for** loop is handy because the code in the loop can access the index (as the variable **i**) and the value at that

Introduction to Python Programming

index (as `supplies[i]`). Best of all, `range(len(supplies))` will iterate through all the indexes of `supplies`, no matter how many items it contains.

The `in` and `not in` Operators

- You 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: a value to look for in a list and the list where it may be found.*
- These expressions will evaluate to a **Boolean value**.

Example:

```
>>> 'baby' in ['hello', 'hi', 'baby', 'thejas']
True
>>> spam = ['hello', 'hi', 'baby', 'thejas']
>>> 'cat' not in spam
True
>>> 'cat' in spam
False
>>> 'baby' not in spam
False
>>> 'baby' in spam
True
```

For example,

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.

```
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.')
```

The output

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

The Multiple Assignment Trick

The *multiple assignment tricks* are a shortcut that lets you *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]
```

you could type this line of code:

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

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#37>", line 1, in <module>
    size, color, disposition, name = cat
ValueError: not enough values to unpack (expected 4, got 3)
```

Augmented Assignment Operators

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

For example, after assigning 45 to the variable `sum`, you would increase the value in `sum` by 1 with the following code.

Example:

```
>>> sum=45
>>> sum=sum+1
>>> sum
46
```

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

Example:

```
>>> sum=45
>>> sum+=1
>>> sum
46
```

There are augmented assignment operators for the `+`, `-`, `*`, `/`, and `%` operators, described in the below Table-2.1.

Table-2.1: The Augmented Assignment Operators.

Augmented assignment statement	Equivalent assignment statement
<code>spam += 1</code>	<code>spam = spam + 1</code>
<code>spam -= 1</code>	<code>spam = spam - 1</code>
<code>spam *= 1</code>	<code>spam = spam * 1</code>
<code>spam /= 1</code>	<code>spam = spam / 1</code>
<code>spam %= 1</code>	<code>spam = spam % 1</code>

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

Example:

```
>>> spam='Hello'
>>> spam+=' World'
>>> spam
'Hello World'
>>> fruit=['Mango']
>>> fruit*= 3
>>> fruit
['Mango', 'Mango', 'Mango']
```

Methods

A **method** is same as a function, except it is “called on” a value.

For **example**,

- If a list value were stored in **fruit**, you would call the **index()** *list method* on that list like: **fruit.index('apple')**. 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.

1. 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 is not in the list, then Python produces a **ValueError** error.

Example:

```
>>> fruit = ["apple", "banana", "orange"]
>>> fruit.index('apple')
0
>>> fruit.index('banana')
1
>>> fruit.index('orange')
2
>>> fruit.index('cherry')
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    fruit.index('cherry')
ValueError: 'cherry' is not in list
```

- When there are duplicates of the value in the list, the index of its **first appearance** is returned. Notice that in the below example, **index()** returns 1, not 3:

Example:

```
>>> fruit = ["apple", "banana", "orange", "banana"]
>>> fruit.index('banana')
1
```


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

- To *add new values to a list*, use the *append()* and *insert()* methods.
- Example to call the *append()* method on a list value stored in the variable `fruit`:

Example:

```
>>> fruit = ["apple", "banana", "orange"]
>>> fruit.append('cherry')
>>> fruit
['apple', 'banana', 'orange', 'cherry']
```

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

```
>>> fruit = ["apple", "banana", "orange"]
>>> fruit.insert(1, 'cherry')
>>> fruit
['apple', 'cherry', 'banana', 'orange']
```

3. Removing Values from Lists with *remove()*

- The *remove()* method is **used to delete the specified value** from the list.

Example:

```
>>> fruit = ["apple", "banana", "orange", "cherry"]
>>> fruit.remove('banana')
>>> fruit
['apple', 'orange', 'cherry']
```

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

Example:

```
>>> fruit = ["apple", "banana", "orange"]
>>> fruit.remove('cherry')
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    fruit.remove('cherry')
ValueError: list.remove(x): x not in list
```

Introduction to Python Programming

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

Example:

```
>>> fruit = ["apple", "banana", "orange", "apple", "cherry"]
>>> fruit
['apple', 'banana', 'orange', 'apple', 'cherry']
>>> fruit.remove('apple')
>>> fruit
['banana', 'orange', 'apple', 'cherry']
```

Note: 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.

4. Sorting the Values in a List with the **sort()** Method

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

Example:

```
>>> num = [2, 5, 3.14, 1, -7]
>>> num.sort()
>>> num
[-7, 1, 2, 3.14, 5]
>>> fruit = ["apple", "banana", "orange", "mango", "cherry"]
>>> fruit.sort()
>>> fruit
['apple', 'banana', 'cherry', 'mango', 'orange']
```

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

Example:

```
>>> fruit = ["apple", "banana", "orange", "mango", "cherry"]
>>> fruit.sort(reverse=True)
>>> fruit
['orange', 'mango', 'cherry', 'banana', 'apple']
```

There are **three things** you should note about the **sort()** method.

1. The **sort()** method sorts the list in place; don't try to capture the return value by writing code like **fruit = fruit.sort()**.

Introduction to Python Programming

2. **Cannot sort** lists that have **both number values** and **string values** in them, since Python doesn't know how to compare these values, it *gives error*.

Example:

```
>>> spam=[1,3,5,'ravi','manju','manu']
>>> spam.sort()
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    spam.sort()
TypeError: '<' not supported between instances of 'str' and 'int'
```

3. ***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:

```
>>> fruit = ['Apple', 'apple', 'Banana', 'banana', 'Cherry', 'cherry']
>>> fruit.sort()
>>> fruit
['Apple', 'Banana', 'Cherry', 'apple', 'banana', 'cherry']
```

- If you 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.

Example Program: Magic 8 Ball with a List

Using lists, you can write a much more elegant version of the First Module; (Example for Return Values and return Statements) Magic 8 Ball program.

Instead of several lines of nearly identical **elif** statements, you can create a **single list** that the code works with.

```
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)])
```

OUTPUT

```
Reply hazy try again
Yes definitely
Very doubtful
```

- Notice 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`**.
- The **benefit of this approach** is that you **can easily add and remove strings to the messages list without changing other lines of code**.

Introduction to Python Programming

List-like Types: Strings and Tuples

- **Strings** and **Lists** are actually similar, if you consider a string to be a “list” of single text characters.
- Many of the things you 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[1]
'o'
>>> name[-1]
'e'
>>> name[0:4]
'Zoph'
>>> Zo in name
Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    Zo in name
NameError: name 'Zo' is not defined
>>> '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 * * *
```

Mutable and Immutable Data Types

But **lists** and **strings** are **different in an important way**. 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#3>", 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**.

- Although a **list value** *is* **mutable**, the **second line in the following code** *does not modify the list* **eggs**:

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

Introduction to Python Programming

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]`.

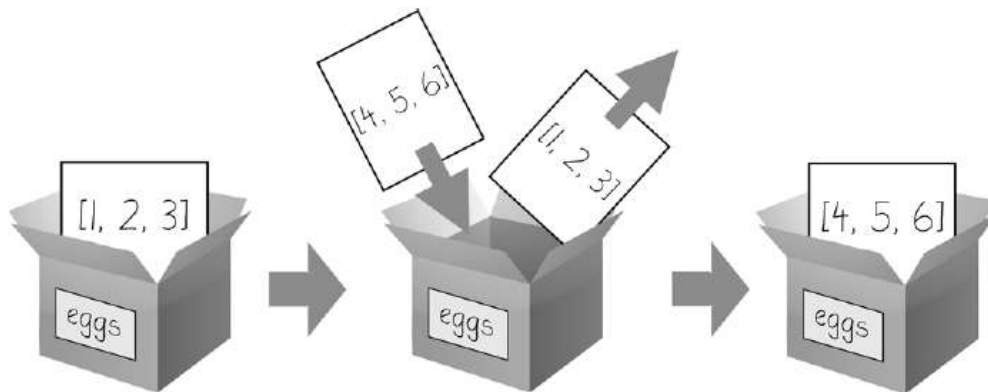


Figure-2.1: When `eggs = [4, 5, 6]` is executed, the contents of `eggs` are replaced with a new list value.

If you wanted to *actually modify the original list in `eggs`* to contain `[4, 5, 6]`, you would **have to do something** like this:

```
>>> eggs = [1, 2, 3]
>>> 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]
```

In the above example, **the list value that `eggs` ends up with is the same list value it started with**. It's just that **this list has been changed**, rather **than overwritten**.

Figure 2.2: Depicts the seven changes made by the first seven lines in the previous example.

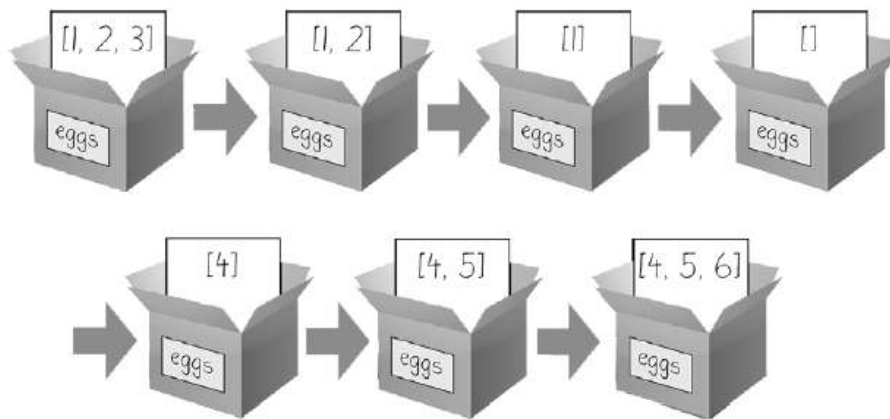


Figure 2-2: The ***del*** statement and the ***append()*** method modify the same list value in place.

- Changing a value of a mutable data type (like what the ***del*** statement and ***append()*** method do in the previous example) changes the value in place, since the variable's value is not replaced with a new list value.
- **Mutable versus immutable types** may seem like a meaningless distinction, but “**Passing References**” on next topics we will explain the different behavior when calling functions with *mutable arguments* versus *immutable arguments*.
- But first, let's find out about the ***tuple data type***, which is ***an immutable*** form of the ***list*** data type.

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].

Example:

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

Introduction to Python Programming

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.

The following example displays the TypeError error message:

Example:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[0]
'hello'
>>> eggs[1]
42
>>> eggs=99
>>> eggs[1]= 99
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    eggs[1]= 99
TypeError: 'int' 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. *Otherwise*, Python will think you've 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, To see the distinction:

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

- We can use tuples to convey to anyone reading your code that **you don't intend for that sequence of values to change**.
- If you need an ordered sequence of values that never changes, *use a tuple*.
- A second benefit of using tuples instead of lists is that, because they are **immutable** and **their contents don't change**, Python can implement some

Introduction to Python Programming

optimizations that make code using tuples slightly faster than code using lists.

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

Just like how `str(50)` will return `'50'`, the string representation of the integer 50, the functions `list()` and `tuple()` will **return** list and tuple versions of the values passed to them.

➤ Notice that the **return value** is of a **different data type** than the value passed:

Example:

```
>>> tuple(['apple','orange',5])
('apple', 'orange', 5)
>>> list(('apple','orange',5))
['apple', 'orange', 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**.

REFERENCES

As you've seen, variables store *strings* and *integer* values.

```
>>> apple=50
>>> orange=apple
>>> apple=100
>>> apple
100
>>> orange
50
```

➤ Assign 50 to the apple variable, and then you copy the value in apple and assign it to the variable orange. When you later change the value in apple to 100, this doesn't affect the value in orange. This is because **apple** and **orange** are **different variables that store different values**.

Introduction to Python Programming

- When you assign a **list** to a **variable**, you 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.
- Here is some code that will make this distinction easier to understand.

Example:

```
1 >>> apple=[0,1,2,3,4,5]
2 >>> orange=apple
3 >>> orange[1]='Hello!'
>>> apple
[0, 'Hello!', 2, 3, 4, 5]
>>> orange[2]='Hi!'
>>> apple
[0, 'Hello!', 'Hi!', 3, 4, 5]
>>> orange[4]='Welcome'
>>> apple
[0, 'Hello!', 'Hi!', 3, 'Welcome', 5]
```

The code changed only the orange list, but it seems that both the orange and apple lists have changed.

- When you create the **list** 1, you assign a reference to it in the **apple** variable. But the next line 2 copies only the **list** reference in **apple** to **orange**, *not the list value itself*.
- This means the values stored in **apple** and **orange** now both refer to the **same list**. There is **only one underlying list** because the **list** itself was never actually copied.
- So when you modify the first element of **orange** 3, you are modifying the same **list** that **orange** refers to.

Remember that variables are like boxes that contain values. Figure 2-3 shows what happens when a list is assigned to the apple variable.

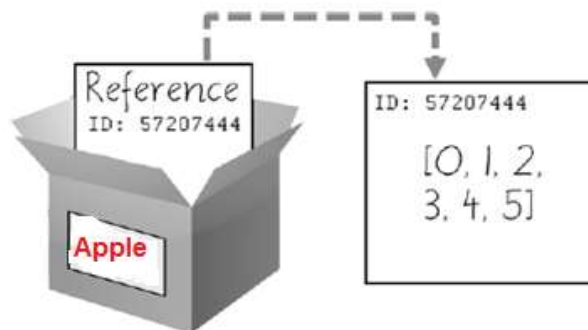


Figure 2-3: Apple = [0, 1, 2, 3, 4, 5] stores a reference to a list, not the actual list.

Introduction to Python Programming

- In Figure 2-4, the reference in `apple` is copied to `orange`. Only a new reference was created and stored in `orange`, not a new list. Note how both references refer to the same list.

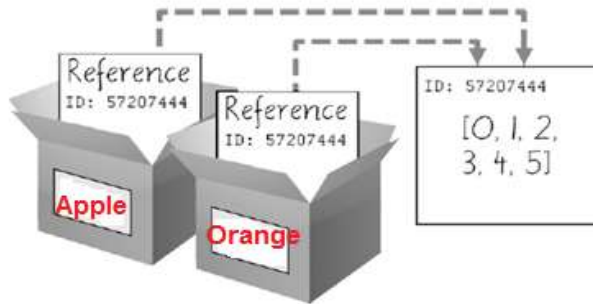


Figure 2-4: `apple = orange` copies the reference, not the list.

When you alter the list that `orange` refers to, the list that `apple` refers to is also changed, because both `orange` and `apple` refer to the same list. You can see this in Figure 2-5.

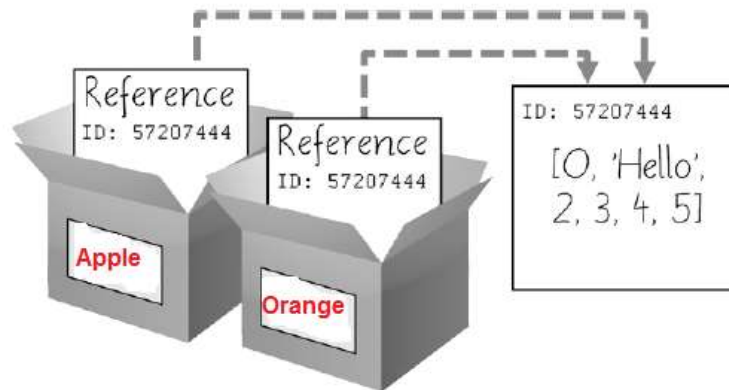


Figure 2-5: `orange[1] = 'Hello!'` modifies the list that both variables refer to.

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

Introduction to Python Programming

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.

```
def eggs(someParameter):  
    someParameter.append('Hello')  
  
spam = [1, 2, 3]  
eggs(spam)  
print(spam)
```

OUTPUT

[1, 2, 3, 'Hello']

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.

The output of the above program is: [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.

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

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']
```

Introduction to Python Programming

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**.

As you can see in **Figure 2-6**, the **reference ID numbers** are **no longer the same** for **both variables** because the **variables refer to independent lists**.

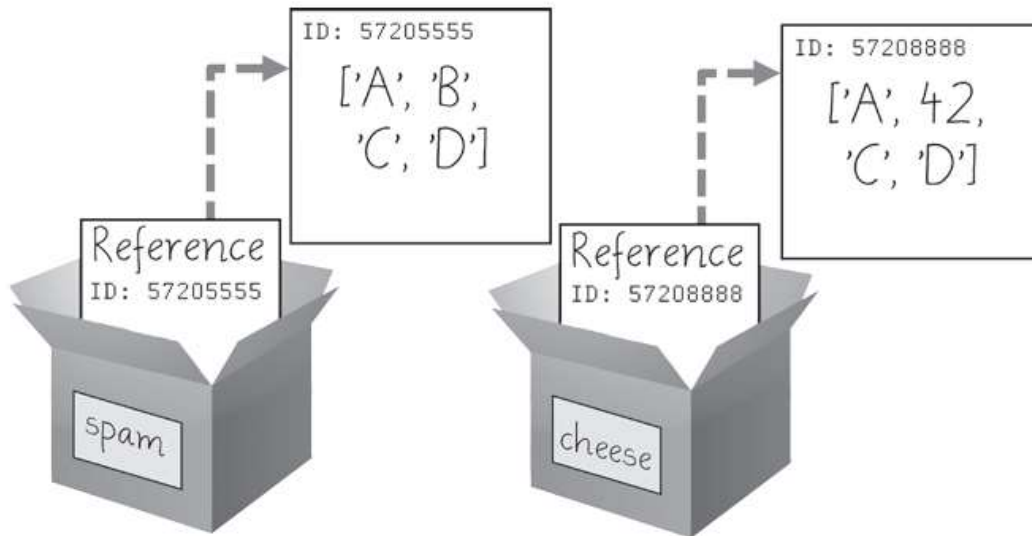


Figure 2-6: **cheese = copy.copy(spam)** creates a **second list** that can be modified independently of the first.

- 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.

```
>>> import copy
>>> p=[1,[2,3],[4,8,2],9,[6,4]]
>>> p
[1, [2, 3], [4, 8, 2], 9, [6, 4]]
>>> q=copy.deepcopy(p)
>>> q
[1, [2, 3], [4, 8, 2], 9, [6, 4]]
>>> q[2]=10
>>> q
[1, [2, 3], 10, 9, [6, 4]]
>>> q[4]=[3,4,5,6]
>>> q
[1, [2, 3], 10, 9, [3, 4, 5, 6]]
```

Dictionaries and Structuring Data

The Dictionary Data Type

- Like a list, a **dictionary** is a **collection of many values**. But **unlike indexes for lists**, **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**. In code, 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.

```
>>> myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
>>> myCat['size']
'fat'
>>> myCat['color']
'gray'
>>> myCat['disposition']
'loud'
>>> 'My cat has ' + myCat['color'] + ' fur.'
'My cat has gray fur.'
>>> myCat = {'size': '50', 'color': 'gray', 'disposition': 'loud'}
>>> myCat['size']
'50'
```

- **Dictionaries can still use integer values as keys**, just **like lists use integers** for **indexes**, 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
```

- Because ***dictionaries are not ordered, they can't be sliced like lists***.
- 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.
- Notice the error message that shows up because there is no **'color'** key:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> spam['color']
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    spam['color']
KeyError: 'color'
```

- Though dictionaries are not ordered, the fact that **you can have arbitrary values for the keys** allows you to organize your data in powerful ways.
- If you want to store the data about your **friends' birthdays**. You can use a **dictionary with the *names as keys* and the *birthdays as values***.

Introduction to Python Programming

```
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.')
```

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

The output of the above code is.

```
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)
```

All the data you enter in this program is forgotten when the program terminates.

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

- There are 3 dictionary methods that will return list-like values of the dictionary's keys, values, or both keys and values: **keys()**, **values()**, and **items()**.
- The **values returned by these methods are not true lists**:
- They **cannot be modified** and **do not have an **append()** method**. But these data types (**dict_keys**, **dict_values**, and **dict_items**, respectively) *can be used in **for** loops*.

To see how these methods work,

```
>>> spam = {'color': 'red', 'age': 50}
>>> for v in spam.values():
    print(v)
```

```
red
50
```

Here, a **for** loop iterates over each of the values in the spam dictionary. A for loop can also iterate over the keys or both keys and values:

Example:

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

```
color
age
```

```
>>> for i in spam.items():
    print(i)
```

```
('color', 'red')
('age', 50)
```

Using the **keys()**, **values()**, and **items()** methods, a **for** loop can iterate over the **keys**, **values**, or **key-value pairs** in a *dictionary*, respectively. Notice that the values in the **dict_items** value returned by the **items()** method are tuples of the key and value.

Introduction to Python Programming

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

Example:

```
>>> 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']**. You 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))
```

Example:

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

Checking Whether a Key or Value Exists in a Dictionary

- Recall from the previous module that the **in** and **not in** operators can **check whether a value exists in a list**.
- You can also use these operators to see whether a certain **key** or **value exists** in a dictionary.

Example:

```
>>> 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
```

Introduction to Python Programming

- In the previous example, notice that **'color' in spam** is essentially a **shorter version** of writing **'color' in spam.keys()**.
 - **This is always the case:** If you ever want to check whether a **value** is (or isn't) a **key** in the dictionary, you can simply use the **in** (or **not in**) **keyword** with the dictionary **value** itself.
-

The `get()` Method

- It's tedious to check whether a key exists in a dictionary before accessing that key's value.
- Fortunately, dictionaries have a **`get()`** method that takes two arguments:
 1. The **key** of the value to retrieve.
 2. A **fallback value** to return if that **key** does not exist.

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

- Because there is no **'eggs' key** in the **Items** dictionary, the **default value 0** is returned by the **`get()`** method.
- Without using **`get()`**, the code would have caused an **error message**.

Example:

```
>>> Items = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(Items['eggs']) + ' eggs.'
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    'I am bringing ' + str(Items['eggs']) + ' eggs.'
KeyError: 'eggs'
```

The.setdefault() Method

You have to **set a value** in a dictionary for a certain **key** only if that **key** does not already have a value.

Example:

```
>>> 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.

- The **first argument** passed to the **method** is the **key** to check for.
- 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
{'name': 'Pooka', 'age': 5, 'color': 'black'}
>>> spam.setdefault('size', 'fat')
'fat'
>>> spam
{'name': 'Pooka', 'age': 5, 'color': 'black', 'size': 'fat'}
```

- Below, the **first time** **setdefault()** is called, the **dictionary in spam** changes to **{ 'name': 'Pooka', 'age': 5, 'color': 'black' }**. 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'**.

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

- The **setdefault()** method is a **nice shortcut** to ensure that a **key exists**.

Introduction to Python Programming

Here is a **short 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. When you run this program.

The **output** will look like this:

```
{ ' ': 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 }
```

- From the output, you can see that the lowercase letter **c** appears **3** times, the **space character** appears **13 times**, and the uppercase letter **A** appears **1** time. This program **will work no matter what string is inside the message variable**, even if the string is millions of characters long!
-

Pretty Printing

- If you **import** the ***pprint*** module into your programs, you'll have access to the ***pprint()*** and ***pformat()*** functions that will “**pretty print**” a dictionary's values.
- This is helpful when you want a **cleaner display of the items in a dictionary** than what ***print()*** provides.

```
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)
```

This time, when the program is run, the **output looks** much cleaner, with the keys sorted.

```
{' ': 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}
```

The ***pprint.pprint()*** function is especially helpful when the **dictionary itself contains nested lists or dictionaries**.

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

These two lines are equivalent to each other:

pprint.pprint(someDictionaryValue)

print(pprint.pformat(someDictionaryValue))

USING DATA STRUCTURES TO MODEL REAL-WORLD THINGS

- Even before the Internet, it was possible to play a game of chess with someone on the other side of the world.
- Each player would set up a chessboard at their home and then take turns mailing a postcard to each other describing each move.
- To do this, the players needed a way to unambiguously describe the state of the board and their moves.
- In **algebraic chess notation**, the **spaces on the chessboard** are identified by a **number** and **letter coordinate**, as in Figure X.1.

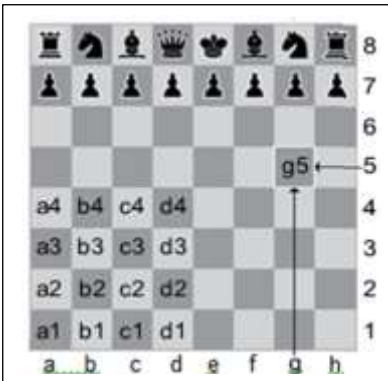


Figure X.1: The coordinates of A chessboard in algebraic chess notation

- The chess pieces are identified by letters: **K** for king, **Q** for queen, **R** for rook, **B** for bishop, and **N** for knight.
- Describing a move uses the letter of the piece and the coordinates of its destination.
- A pair of these moves describes what happens in a single turn (with white going first);

For instance, the notation **2. Nf3 Nc6** indicates that **white** moved a **knight** to **f3** and **black** moved a **knight** to **c6** on the **second turn** of

- Here, the point is that you can use it to unambiguously describe a game of chess without needing to be in front of a chessboard.
- Your opponent can even be on the other side of the world!
- In fact, you don't even need a physical chess set if you have a good memory:
 - You can just read the mailed chess moves and update boards you have in your imagination.
- As computers have good memories, a program on a modern computer can easily store billions of strings like '**2. Nf3 Nc6**'.
- This is how computers can play chess without having a physical chessboard.
- They model data to represent a chessboard, and you can write code to work with this model.
- This is where lists and dictionaries can come in.
- You can use them to model real-world things, like chessboards.
- For the first example, you'll use a game that's a little simpler than chess: **tic-tac-toe**.

A Tic-Tac-Toe Board

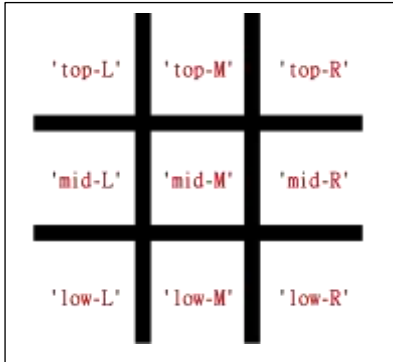


Figure X.2: The slots of a tic-tac-toe board with their corresponding keys

- 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, you can assign **each slot** a **string-value key**, as shown in Figure X-2.
- You can use string values to represent what's in each slot on the board: '**X**', '**O**', or ' ' (a **space** character).
- Thus, you'll need to store **nine strings**. You can use a dictionary of values for this.
- The string values with the keys '**top-R**', '**low-L**', '**mid-M**' can represent **top-right corner**, **low-left corner** and **middle** respectively and so on.

- This **dictionary** is a **data structure** that represents a **tic-tac-toe board**.
- Store this **board-as-a-dictionary** in a variable named **theBoard**.
- Enter the following source code, saving it as **ticTacToe.py**:

```
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-tac-toe board** in Figure X.3.

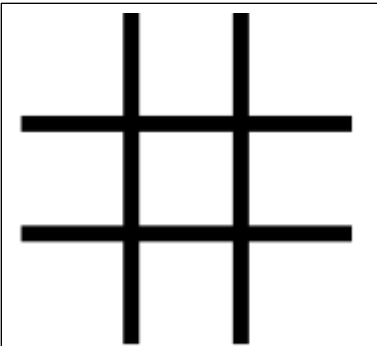


Figure X.3: An empty tic-tac-toe

- 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 figure X.4

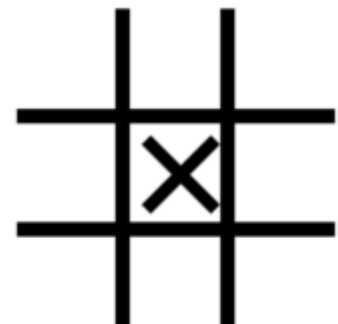


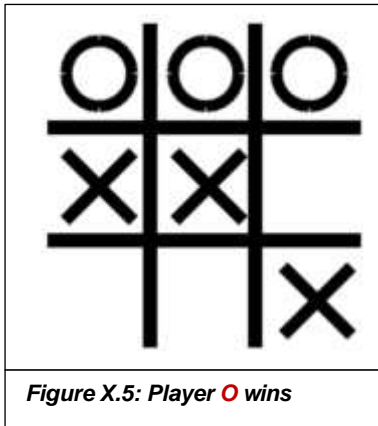
Figure X.4: The first move

- 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'}
```

Introduction to Python Programming

- The data structure in **theBoard** now represents the **tic-tac-toe board** in Figure X.5.



- Let's create a function to print the board dictionary onto the screen. Make the following addition to *ticTacToe.py* (new code is in bold):

```
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)
```

- When you run this program, **printBoard()** will print out a blank **tic-tac- toe board** like:

```
| |  
-+-+-  
| |  
-+-+-  
| |
```

Introduction to Python Programming

- The **printBoard()** function can handle any **tic-tac-toe data structure** you pass it. Try changing the code to the following:

```
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)
```

- Now when you run this program, the new board will be printed to the screen.

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

- Because you created a **data structure** to represent a **tic-tac-toe board** and wrote code in **printBoard()** to interpret that data structure, you now have a program that *“models”* the **tic-tac-toe board**.
- The **printBoard()** function expects the **tic-tac-toe data structure** to be a **dictionary with keys for all nine slots**. If the *dictionary you passed was missing*, say, the **'mid-L'** key, your program would *no longer work*.

```
O|O|O
```

```
-+-+-
```

```
Traceback (most recent call last):
```

```
File "ticTacToe.py", line 10, in <module>
```

```
    printBoard(theBoard)
```

```
File "ticTacToe.py", line 6, in printBoard
```

```
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
```

```
KeyError: 'mid-L'
```

Introduction to Python Programming

Now let's add code that allows the players to enter their moves.

Modify the *ticTacToe.py* program to look like this:

```
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):  
1      printBoard(theBoard)  
        print('Turn for ' + turn + '. Move on which space?')  
2      move = input( )  
3      theBoard[move] = turn  
4      if turn == 'X': turn = 'O'  
        else:  
            turn = 'X'  
    printBoard(theBoard)
```

- 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.
- When you run this program, like follows:

```
||  
-+-+-
```

```
||  
-+-+-
```

```
||
```

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
```

-
- This isn't a complete **tic-tac-toe** game—for instance, it doesn't ever check whether a player has won—but it's enough to see how data structures can be used in programs.

Introduction to Python Programming

Nested Dictionaries and Lists

- Modeling a **tic-tac-toe board** was fairly simple:
- The board needed only a single dictionary value with nine key-value pairs.
- As you model more complicated things, you may need dictionaries and lists that contain other dictionaries and lists.
- Lists are useful to contain an ordered series of values, and dictionaries are useful for associating keys with values.
- For example, here's a program that uses a dictionary that contains other dictionaries in order to see who is bringing what to a picnic.
- The **totalBrought()** function can read this data structure and calculate the total number of items being brought by all the guests.

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

```
def totalBrought(guests, item):  
    numBrought = 0  
  
    1    for k, v in guests.items():  
    2        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 **key-value** 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, its 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**.

Introduction to Python Programming

- The **output** of this program looks like this:

Number of things being brought:

- Apples 7
- Cups 3
- Cakes 0
- Ham Sandwiches 3
- Apple Pies 1

- This may seem like such a simple thing to model that you wouldn't need to bother with writing a program to do it.
- But realize that this same **totalBrought()** function could easily handle a dictionary that contains thousands of guests, each bringing *thousands* of different picnic items.
- Then having this information in a data structure along with the **totalBrought()** function would save you a lot of time!
- You can model things with data structures in whatever way you like, as long as the rest of the code in your program can work with the data model correctly.

-----XXXXXX-----