LISTS

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

The List Data Type

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. Items are separated with commas (that is, they are comma-delimited).

In [2]:
```
[1, 2, 3]
```

Out[2]:
```
[1, 2, 3]
```

In [3]:
```
['cat', 'bat', 'rat', 'elephant']
```

Out[3]:
```
['cat', 'bat', 'rat', 'elephant']
```

In [4]:
```
['hello', 3.1415, True, None, 42]
```

Out[4]:
```
['hello', 3.1415, True, None, 42]
```

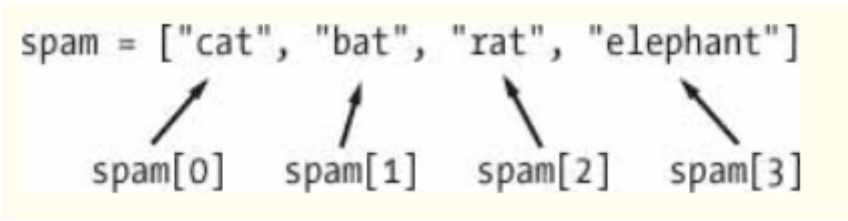
In [5]:
```
spam = ['cat', 'bat', 'rat', 'elephant']
```

In [6]:
```
spam
```

Out[6]:
```
['cat', 'bat', 'rat', 'elephant']
```

Getting Individual Values in a List with Indexes

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.



In [7]:

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

In [8]:

```
spam[0]
```

Out[8]:

```
'cat'
```

In [9]:

```
spam[1]
```

Out[9]:

```
'bat'
```

In [10]:

```
spam[2]
```

Out[10]:

```
'rat'
```

In [11]:

```
['cat', 'bat', 'rat', 'elephant'][3]
```

Out[11]:

```
'elephant'
```

In [12]:

```
'Hello, ' + spam[0]
```

Out[12]:

```
'Hello, cat'
```

In [13]:

```
'The ' + spam[1] + ' ate the ' + spam[0] + '.'
```

Out[13]:

```
'The bat ate the cat.'
```

In [14]:

```
spam[10000]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-14-3b15763110c6> in <module>
----> 1 spam[10000]

IndexError: list index out of range
```

In [15]:

```
spam[1.0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-a283f69a570a> in <module>
----> 1 spam[1.0]

TypeError: list indices must be integers or slices, not float
```

In [16]:

```
spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
```

In [17]:

```
spam[0]
```

Out[17]:

```
['cat', 'bat']
```

In [18]:

```
 spam[0][1]
```

Out[18]:

```
'bat'
```

Negative Indexes

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

the value -2 refers to the second-to-last index in a list, and so on.

In [19]:

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

In [20]:

```
spam[-1]
```

Out[20]:

```
'elephant'
```

In [21]:

```
spam[-3]
```

Out[21]:

```
'bat'
```

Getting a List from Another List with Slices

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

Notice the difference between indexes and slices.

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

In [22]:

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

In [23]:

```
spam[0:4]
```

Out[23]:

```
['cat', 'bat', 'rat', 'elephant']
```

In [24]:

```
spam[1:3]
```

Out[24]:

```
['bat', 'rat']
```

In [25]:

```
spam[0:-1]
```

Out[25]:

```
['cat', 'bat', 'rat']
```

In [26]:

```
spam[:2]
```

Out[26]:

```
['cat', 'bat']
```

In [27]:

```
spam[1:]
```

Out[27]:

```
['bat', 'rat', 'elephant']
```

In [28]:

```
spam[:]
```

Out[28]:

```
['cat', 'bat', 'rat', 'elephant']
```

In [29]:

```
len(spam)
```

Out[29]:

```
4
```

Changing Values in a List with Indexes

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

In [30]:

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

In [31]:

```
spam[1] = 'aardvark'
```

In [32]:

```
spam
```

Out[32]:

```
['cat', 'aardvark', 'rat', 'elephant']
```

In [33]:

```
spam[2] = spam[1]
```

In [34]:

```
spam
```

Out[34]:

```
['cat', 'aardvark', 'aardvark', 'elephant']
```

In [35]:

```
spam[-1] = 12345
```

In [36]:

```
spam
```

Out[36]:

```
['cat', 'aardvark', 'aardvark', 12345]
```

List Concatenation and List Replication

Lists can be concatenated and replicated just like strings. The + operator combines two lists to create a new list value
and the * operator can be used with a list and an integer value to replicate the list.

In [37]:

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

Out[37]:

```
[1, 2, 3, 'A', 'B', 'C']
```

In [38]:

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

Out[38]:

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

In [39]:

```
spam = [1, 2, 3]
```

In [40]:

```
spam = spam + ['A', 'B', 'C']
```

In [41]:

```
spam
```

Out[41]:

```
[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. All of the values in the list after the deleted value will be
moved up one index.

In [42]:

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

In [43]:

```
del spam[2]
```

In [44]:

```
spam
```

Out[44]:

```
['cat', 'bat', 'elephant']
```

In [45]:

```
del spam[2]
```

In [46]:

```
spam
```

Out[46]:

```
['cat', 'bat']
```

Working with Lists

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

In [47]:

```
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)
```

```
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.):
Lady Macbeth
Enter the name of cat 4 (Or enter nothing to stop.):
Miss Cleo
Enter the name of cat 5 (Or enter nothing to stop.):
nothing
Enter the name of cat 6 (Or enter nothing to stop.):

The cat names are:
  Zophie
  Pooka
  Lady Macbeth
  Miss Cleo
  nothing
```

Using for Loops with Lists

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

In [48]:

```
supplies = ['pens', 'staplers', 'flamethrowers', 'binders']
```

In [49]:

```
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: flamethrowers
Index 3 in supplies is: binders
```

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.

In [50]:

```
'howdy' in ['hello', 'hi', 'howdy', 'heyas']
```

Out[50]:

```
True
```

In [51]:

```
spam = ['hello', 'hi', 'howdy', 'heyas']
```

In [52]:

```
 'cat' in spam
```

Out[52]:

```
False
```

In [53]:

```
'howdy' not in spam
```

Out[53]:

```
False
```

In [54]:

```
 'cat' not in spam
```

Out[54]:

```
True
```

In [55]:

```
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.')
```

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

The Multiple Assignment Trick

The multiple assignment trick (technically called tuple unpacking) is a shortcut that lets you assign multiple variables
with the values in a list in one line of code.

In [56]:
```
cat = ['fat', 'gray', 'loud']
```

In [57]:
```
size = cat[0]
```

In [58]:
```
color = cat[1]
```

In [59]:
```
disposition = cat[2]
```

In [60]:
```
cat = ['fat', 'gray', 'loud']
```

In [61]:
```
size, color, disposition = cat
```

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

In [62]:
```
cat = ['fat', 'gray', 'loud']
```

In [63]:
```
size, color, disposition, name = cat
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-63-1149c93a978e> in <module>
----> 1 size, color, disposition, name = cat

ValueError: not enough values to unpack (expected 4, got 3)
```

Using the enumerate() Function with Lists

Instead of using the range(len(someList)) technique with a for loop to obtain the integer index of the items in the list,
you can call the enumerate() function instead.

On each iteration of the loop, enumerate() will return two values: the index of the item in the list, and the item in the list itself.

In [65]:

```
supplies = ['pens', 'staplers', 'flamethrowers', 'binders']
```

In [66]:

```
for index, item in enumerate(supplies):
    print('Index ' + str(index) + ' in supplies is: ' + item)
```

```
Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flamethrowers
Index 3 in supplies is: binders
```

Using the random.choice() and random.shuffle() Functions with Lists

The random module has a couple functions that accept lists for arguments.
The random.choice() function will return a randomly selected item from the list.

In [2]:

```
import random

pets = ['Dog', 'Cat', 'Moose']
```

In [3]:

```
random.choice(pets)
```

Out[3]:

```
'Cat'
```

In [4]:

```
random.choice(pets)
```

Out[4]:

```
'Moose'
```

In [70]:

```
random.choice(pets)
```

Out[70]:

```
'Cat'
```

In [71]:

```
random.choice(pets)
```

Out[71]:

```
'Moose'
```

In [72]:

```
random.choice(pets)
```

Out[72]:

```
'Dog'
```

The random.shuffle() function will reorder the items in a list. This function modifies the list in place, rather than returning a new list.

In [5]:

```
import random

people = ['Alice', 'Bob', 'Carol', 'David']
```

In [10]:

```
random.shuffle(people)
```

In [11]:

```
people
```

Out[11]:

```
['David', 'Carol', 'Bob', 'Alice']
```

In [9]:

```
random.shuffle(people)
```

In [77]:

```
people
```

Out[77]:

```
['Carol', 'Bob', 'David', 'Alice']
```

Augmented Assignment Operators

When assigning a value to a variable, you will frequently use the variable itself. For example, after assigning 42

to the variable spam,
you would increase the value in spam by 1 with the following code:

In [78]:

```
spam = 42
```

In [79]:

```
spam = spam + 1
```

In [80]:

```
spam
```

Out[80]:

```
43
```

There are augmented assignment operators for the +, -, *, /, and % operators, described in Table 4-1.

Table 4-1: The Augmented Assignment Operators

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

In [2]:

```
spam = 'Hello,'
```

In [3]:

```
spam += ' world!'
```

In [4]:

```
spam
```

Out[4]:

```
'Hello, world!'
```

In [84]:

```
bacon = ['Zophie']
```

In [85]:

```
bacon *= 3
```

In [86]:

```
bacon
```

Out[86]:

```
['Zophie', 'Zophie', 'Zophie']
```

Methods

A method is the same thing as a function, except it is “called on” a value. For example, if a list value were stored in spam,
you would call the index() list method (which I’ll explain shortly) on that list like so: spam.index('hello').
The method part comes after the value, separated by a period.

Each data type has its own set of methods. The list data type, for example, has several useful methods for finding, adding,
removing, and otherwise manipulating values in a list.

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.

In [12]:

```
spam = ['hello', 'hi', 'howdy', 'heyas']
```

In [88]:

```
spam.index('hello')
```

Out[88]:

```
0
```

In [89]:

```
spam.index('heyas')
```

Out[89]:

```
3
```

In [90]:

```
spam.index('howdy howdy howdy')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-90-aa717b951a7e> in <module>
----> 1 spam.index('howdy howdy howdy')

ValueError: 'howdy howdy howdy' is not in list
```

In [91]:

```
spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
```

In [92]:

```
spam.index('Pooka')
```

Out[92]:

```
1
```

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

To add new values to a list, use the append() and insert() methods.

In [13]:

```
spam = ['cat', 'dog', 'bat']
```

In [14]:

```
spam.append('moose')
```

In [15]:

```
spam
```

Out[15]:

```
['cat', 'dog', 'bat', 'moose']
```

In [16]:

```
spam = ['cat', 'dog', 'bat']
```

In [97]:

```
spam.insert(1, 'chicken')
```

In [98]:

```
spam
```

Out[98]:

```
['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.

In [99]:

```
eggs = 'hello'
```

In [100]:

```
eggs.append('world')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-100-a21e47a7597b> in <module>
----> 1 eggs.append('world')

AttributeError: 'str' object has no attribute 'append'
```

In [101]:

```
bacon = 42
```

In [102]:

```
bacon.insert(1, 'world')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-102-c05641e45155> in <module>
----> 1 bacon.insert(1, 'world')

AttributeError: 'int' object has no attribute 'insert'
```

Removing Values from Lists with the remove() Method

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

In [103]:

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

In [104]:

```
spam.remove('bat')
```

In [105]:

```
spam
```

Out[105]:

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

In [106]:

```
spam.remove('chicken')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-106-2f6af6627885> in <module>
----> 1 spam.remove('chicken')

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

In [107]:

```
spam = ['cat', 'bat', 'rat', 'cat', 'hat', 'cat']
```

In [17]:

```
spam.remove('cat')
```

In [109]:

```
spam
```

Out[109]:

```
['bat', 'rat', 'cat', 'hat', 'cat']
```

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.

In [18]:

```
spam = [2, 5, 3.14, 1, -7]
```

In [19]:

```
spam.sort()
```

In [20]:

```
spam
```

Out[20]:

```
[-7, 1, 2, 3.14, 5]
```

In [113]:

```
spam = ['ants', 'cats', 'dogs', 'badgers', 'elephants']
```

In [114]:

```
spam.sort()
```

In [115]:

```
spam
```

Out[115]:

```
['ants', 'badgers', 'cats', 'dogs', 'elephants']
```

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

In [116]:

```
spam.sort(reverse=True)
```

In [117]:

```
spam
```

Out[117]:

```
['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 capture the return value by writing code like spam = spam.sort().

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

In [118]:

```
spam = [1, 3, 2, 4, 'Alice', 'Bob']
```

In [119]:

```
spam.sort()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-119-9c194a01a31d> in <module>
----> 1 spam.sort()

TypeError: '<' not supported between instances of 'str' and 'int'
```

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

In [120]:

```
spam = ['Alice', 'ants', 'Bob', 'badgers', 'Carol', 'cats']
```

In [121]:

```
spam.sort()
```

In [122]:

```
spam
```

Out[122]:

```
['Alice', 'Bob', 'Carol', 'ants', 'badgers', 'cats']
```

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

In [123]:

```
spam = ['a', 'z', 'A', 'Z']
```

In [124]:

```
spam.sort(key=str.lower)
```

In [125]:

```
spam
```

Out[125]:

```
['a', 'A', 'z', 'Z']
```

Reversing the Values in a List with the reverse() Method

If you need to quickly reverse the order of the items in a list, you can call the reverse() list method.

In [127]:

```
spam = ['cat', 'dog', 'moose']
```

In [128]:

```
spam.reverse()
```

In [129]:

```
spam
```

Out[129]:

```
['moose', 'dog', 'cat']
```

EXCEPTIONS TO INDENTATION RULES IN PYTHON

In most cases, the amount of indentation for a line of code tells Python what block it is in. There are some exceptions to this rule, however. For example, lists can actually span several lines in the source code file. The indentation of these lines does not matter; Python knows that the list is not finished until it sees the ending square bracket. For example, you can have code that looks like this:

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

Of course, practically speaking, most people use Python's behavior to make their lists look pretty and readable, like the messages list in the Magic 8 Ball program.

You can also split up a single instruction across multiple lines using the \ *line continuation character* at the end. Think of \ as saying, "This instruction continues on the next line." The indentation on the line after a \ line continuation is not significant. For example, the following is valid Python code:

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

These tricks are useful when you want to rearrange long lines of Python code to be a bit more readable.

Example Program: Magic 8 Ball with a List

In [2]:

```
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)])
```

```
Very doubtful
```

Sequence Data Types

The Python sequence data types include lists, strings, range objects returned by range(), and tuples.

In [21]:

```
name='Zophie'
```

In [22]:

```
name[0]
```

Out[22]:

```
'Z'
```

In [23]:

```
name[0:4]
```

Out[23]:

```
'Zoph'
```

In [134]:

```
for i in name:
    print('* * * ' + i + ' * * *')
```

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

Mutable and Immutable Data Types

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.

In [135]:

```
name = 'Zophie a cat'
```

In [136]:

```
name[7] = 'the'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-136-a8d518806477> in <module>
----> 1 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.

In [137]:

```
name = 'Zophie a cat'
```

In [138]:

```
newName = name[0:7] + 'the' + name[8:12]
```

In [139]:

```
name
```

Out[139]:

```
'Zophie a cat'
```

In [140]:

```
newName
```

Out[140]:

```
'Zophie the cat'
```

In [24]:

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

In [25]:

```
eggs = [4, 5, 6]
```

In [143]:

```
eggs
```

Out[143]:

```
[4, 5, 6]
```

In [26]:

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

In [27]:

```
del eggs[2]
```

In [146]:

```
del eggs[1]
```

In [147]:

```
del eggs[0]
```

In [148]:

```
eggs.append(4)
```

In [149]:

```
eggs.append(5)
```

In [150]:

```
eggs.append(6)
```

In [151]:

```
eggs
```

Out[151]:

```
[4, 5, 6]
```

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].

In [28]:

```
eggs = ('hello', 42, 0.5)
```

In [153]:

```
eggs[0]
```

Out[153]:

```
'hello'
```

In [154]:

```
eggs[1:3]
```

Out[154]:

```
(42, 0.5)
```

In [155]:

```
len(eggs)
```

Out[155]:

```
3
```

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

In [156]:

```
eggs = ('hello', 42, 0.5)
```

In [157]:

```
eggs[1] = 99
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-157-3c87a10f5e34> in <module>
----> 1 eggs[1] = 99

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

In [158]:

```
type(('hello',))
```

Out[158]:

```
tuple
```

In [159]:

```
type(('hello'))
```

Out[159]:

```
str
```

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

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

In [160]:

```
tuple(['cat', 'dog', 5])
```

Out[160]:

```
('cat', 'dog', 5)
```

In [161]:

```
 list(('cat', 'dog', 5))
```

Out[161]:

```
['cat', 'dog', 5]
```

In [162]:

```
list('hello')
```

Out[162]:

```
['h', 'e', 'l', 'l', 'o']
```

References

As you've seen, variables "store" strings and integer values. However, this explanation is a simplification of what Python is actually doing.
Technically, variables are storing references to the computer memory locations where the values are stored.

In [29]:

```
spam = 42
```

In [30]:

```
 cheese = spam
```

When you assign 42 to the spam variable, you are actually creating the 42 value in the computer's memory and storing a reference to it in the spam variable.

When you copy the value in spam and assign it to the variable cheese, you are actually copying the reference.

In [32]:

```
spam=100
cheese
```

Out[32]:

```
42
```

Both the spam and cheese variables refer to the 42 value in the computer's memory. When you later change the value in spam to 100,
you're creating a new 100 value and storing a reference to it in spam. This doesn't affect the value in cheese. Integers are immutable values that don't change; changing the spam variable is actually making it refer to a completely different value in memory.

But lists don't work this way, because list values can change; that is, lists are mutable.
Here is some code that will make this distinction easier to understand.

In [33]:

```
spam = [0, 1, 2, 3, 4, 5]
```

In [34]:

```
cheese = spam
```

In [35]:

```
cheese[1] = 'Hello!'
```

In [168]:

```
spam
```

Out[168]:

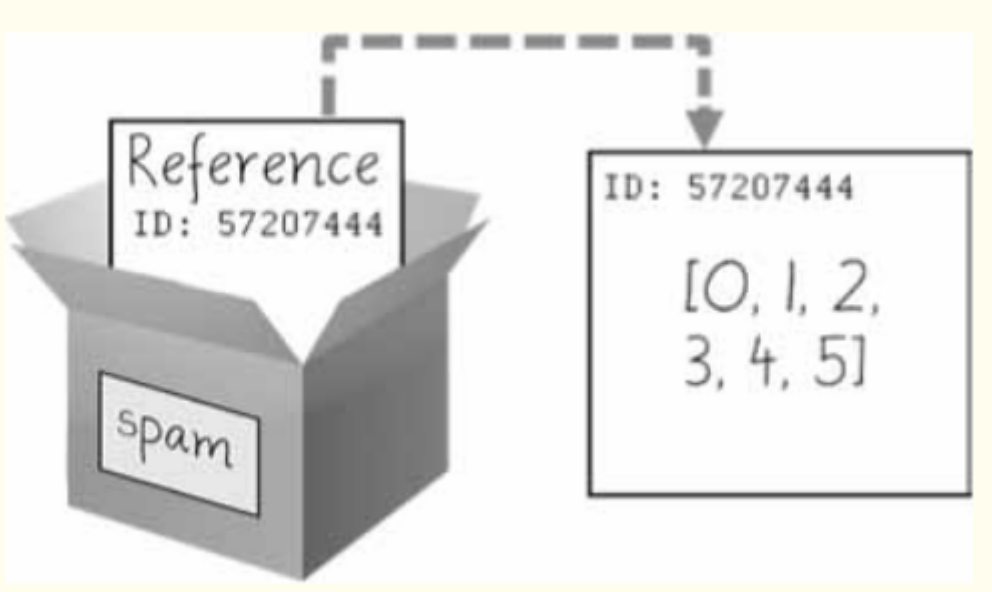
```
[0, 'Hello!', 2, 3, 4, 5]
```

In [169]:

```
cheese
```

Out[169]:

```
[0, 'Hello!', 2, 3, 4, 5]
```



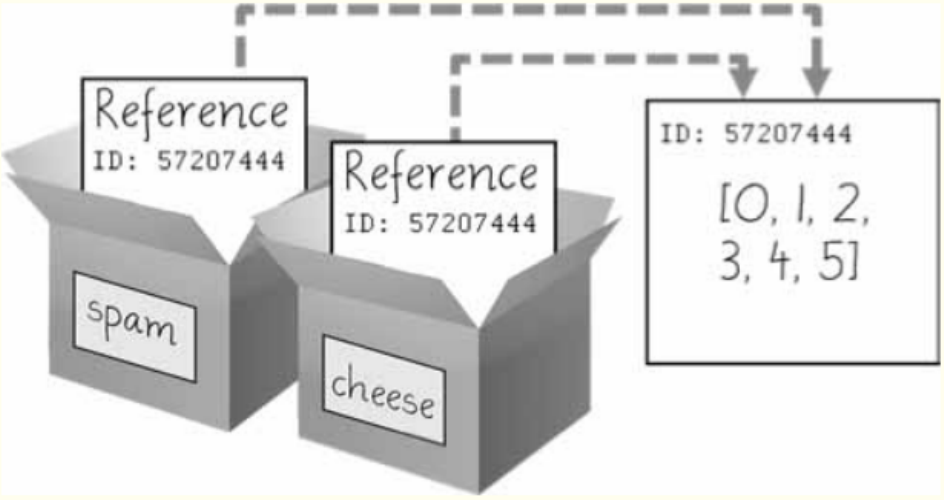


Figure 4-5: spam = cheese copies the reference, not the list.

When you 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. You can see this in Figure 4-6.

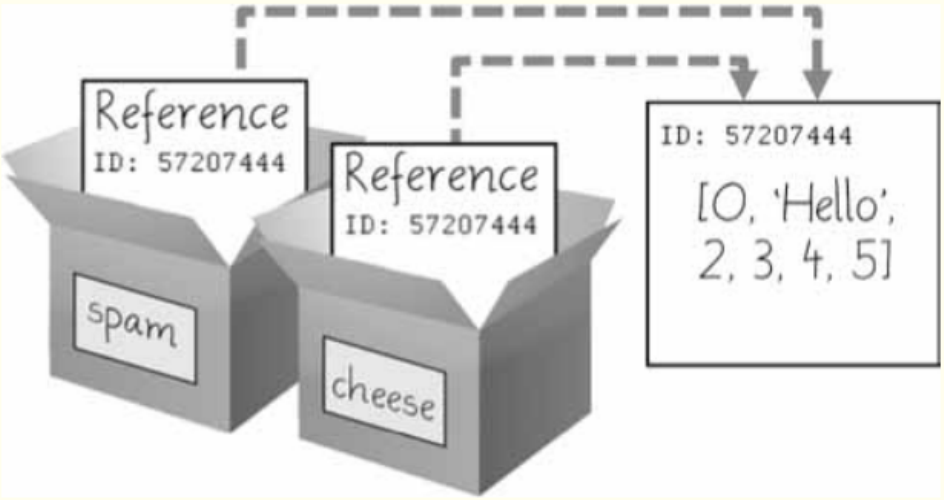


Figure 4-6: cheese[1] = 'Hello!' modifies the list that both variables refer to.

Identity and the id() Function

We can use Python's id() function to understand this. All values in Python have a unique identity that can be obtained with the id() function.

In [170]:

```
id('Howdy')
```

Out[170]:

```
2171417904112
```

In [171]:

```
bacon = 'Hello'
```

In [172]:

```
id(bacon)
```

Out[172]:

```
2171417905392
```

In [173]:

```
bacon += ' world!'
```

In [174]:

```
id(bacon)
```

Out[174]:

```
2171417912752
```

In [175]:

```
bacon
```

Out[175]:

```
'Hello world!'
```

However, lists can be modified because they are mutable objects. The append() method doesn't create a new list object;
it changes the existing list object. We call this "modifying the object in-place."

In [176]:

```
eggs = ['cat', 'dog']
```

In [177]:

```
id(eggs)
```

Out[177]:

```
2171417880520
```

In [178]:

```
eggs.append('moose')
```

In [179]:

```
id(eggs)
```

Out[179]:

```
2171417880520
```

In [180]:

```
eggs = ['bat', 'rat', 'cow']
```

In [181]:

```
id(eggs)
```

Out[181]:

```
2171417844872
```

Python’s automatic garbage collector deletes any values not being referred to by any variables to free up memory.
You don’t need to worry about how the garbage collector works, which is a good thing:
manual memory management in other programming languages is a common source of bugs.

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.

In [36]:

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

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

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

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.

In [37]:

```
import copy
spam = ['A', 'B', 'C', 'D']
```

In [38]:

```
id(spam)
```

Out[38]:

```
2540243631240
```

In [185]:

```
cheese = copy.copy(spam)
```

In [39]:

```
id(cheese)
```

Out[39]:

```
2540244506120
```

In [187]:

```
cheese[1] = 42
```

In [188]:

```
spam
```

Out[188]:

```
['A', 'B', 'C', 'D']
```

In [189]:

```
cheese
```

Out[189]:

```
['A', 42, 'C', 'D']
```

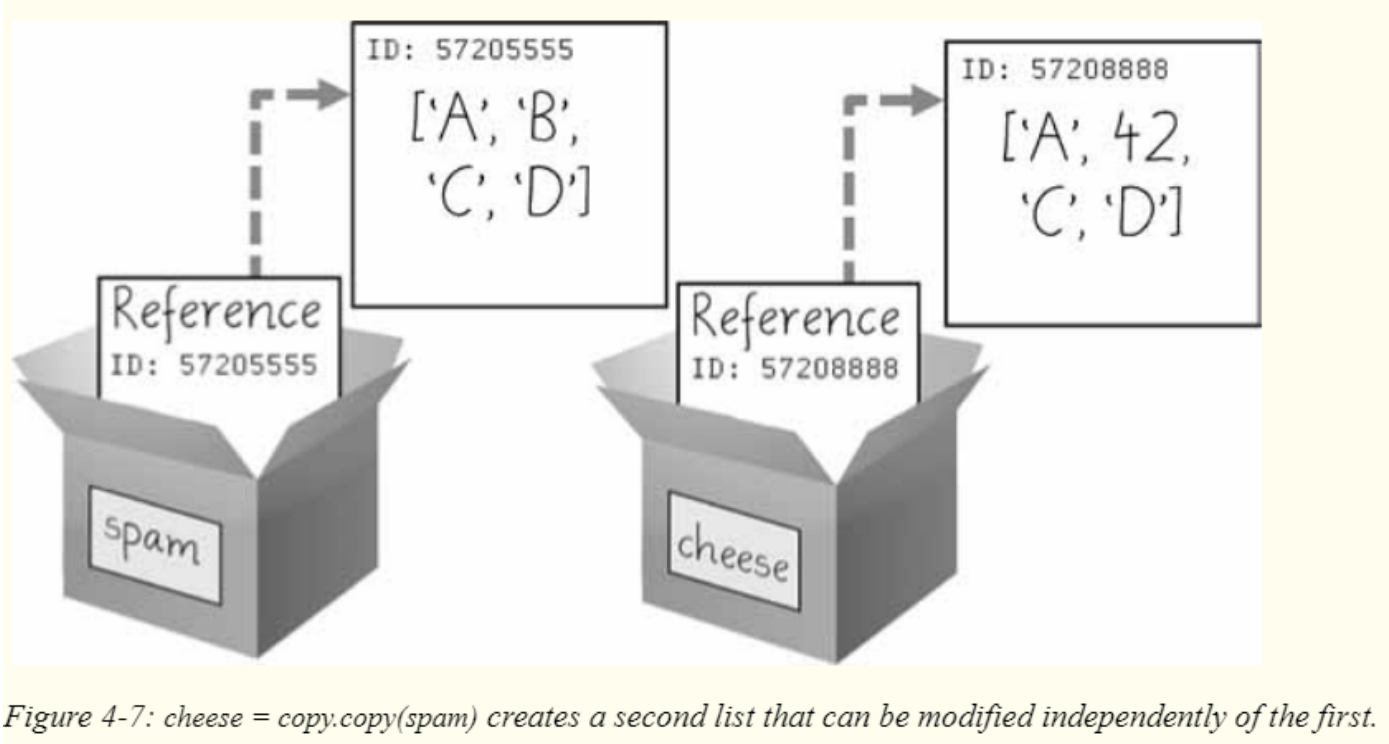


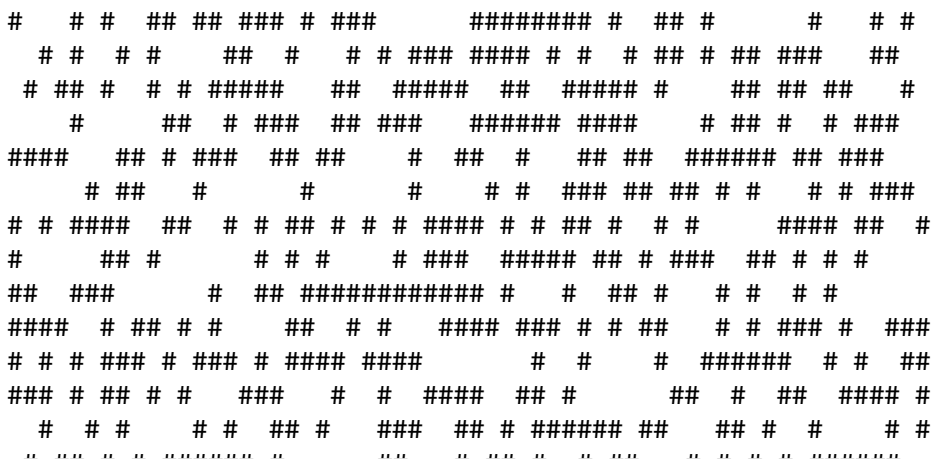
Figure 4-7: cheese = copy.copy(spam) creates a second list that can be modified independently of the first.

A Short Program: Conway's Game of Life

In [190]:

```
# Conway's Game of Life
import random, time, copy
WIDTH = 60
HEIGHT = 20

# Create a list of list for the cells:
nextCells = []
for x in range(WIDTH):
    column = [] # Create a new column.
    for y in range(HEIGHT):
        if random.randint(0, 1) == 0:
            column.append('#') # Add a living cell.
        else:
            column.append(' ') # Add a dead cell.
    nextCells.append(column) # nextCells is a list of column lists.

while True: # Main program loop.
    print('\n\n\n\n\n') # Separate each step with newlines.
    currentCells = copy.deepcopy(nextCells)

    # Print currentCells on the screen:
    for y in range(HEIGHT):
        for x in range(WIDTH):
            print(currentCells[x][y], end='') # Print the # or space.
        print() # Print a newline at the end of the row.

    # Calculate the next step's cells based on current step's cells:
    for x in range(WIDTH):
        for y in range(HEIGHT):
            # Get neighboring coordinates:
            # `% WIDTH` ensures leftCoord is always between 0 and WIDTH - 1
            leftCoord  = (x - 1) % WIDTH
            rightCoord = (x + 1) % WIDTH
            aboveCoord = (y - 1) % HEIGHT
            belowCoord = (y + 1) % HEIGHT

            # Count number of living neighbors:
            numNeighbors = 0
            if currentCells[leftCoord][aboveCoord] == '#':
                numNeighbors += 1 # Top-left neighbor is alive.
            if currentCells[x][aboveCoord] == '#':
                numNeighbors += 1 # Top neighbor is alive.
            if currentCells[rightCoord][aboveCoord] == '#':
                numNeighbors += 1 # Top-right neighbor is alive.
            if currentCells[leftCoord][y] == '#':
                numNeighbors += 1 # Left neighbor is alive.
            if currentCells[rightCoord][y] == '#':
                numNeighbors += 1 # Right neighbor is alive.
            if currentCells[leftCoord][belowCoord] == '#':
                numNeighbors += 1 # Bottom-left neighbor is alive.
            if currentCells[x][belowCoord] == '#':
                numNeighbors += 1 # Bottom neighbor is alive.
            if currentCells[rightCoord][belowCoord] == '#':
                numNeighbors += 1 # Bottom-right neighbor is alive.

            # Set cell based on Conway's Game of Life rules:
            if currentCells[x][y] == '#' and (numNeighbors == 2 or
numNeighbors == 3):
                # Living cells with 2 or 3 neighbors stay alive:
```

```
                nextCells[x][y] = '#'
            elif currentCells[x][y] == ' ' and numNeighbors == 3:
                # Dead cells with 3 neighbors become alive:
                nextCells[x][y] = '#'
            else:
                # Everything else dies or stays dead:
                nextCells[x][y] = ' '
    time.sleep(1) # Add a 1-second pause to reduce flickering.
```



Summary By Quiz

DICTIONARIES AND STRUCTURING DATA

The Dictionary Data Type

A dictionary is a mutable 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 [1]:

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

In [2]:

```
myCat['size']
```

Out[2]:

```
'fat'
```

In [3]:

```
'My cat has ' + myCat['color'] + ' fur.'
```

Out[3]:

```
'My cat has gray fur.'
```

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.

In [4]:

```
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.

In [5]:

```
spam = ['cats', 'dogs', 'moose']
```

In [6]:

```
bacon = ['dogs', 'moose', 'cats']
```

In [7]:

```
spam == bacon
```

Out[7]:

```
False
```

In [8]:

```
eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
```

In [9]:

```
ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
```

In [10]:

```
eggs == ham
```

Out[10]:

```
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.

In [11]:

```
spam = {'name': 'Zophie', 'age': 7}
```

In [12]:

```
spam['color']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-12-4ab2633f52b0> in <module>
----> 1 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.
Say you wanted your program to store data about your friends' birthdays. You can use a dictionary with the names as keys and the birthdays as values.

In [17]:

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

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

    if name in birthdays:
        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()
        birthdays[name] = bday
        print('Birthday database updated.')
```

```
Enter a name: (blank to quit)
mann
I do not have birthday information for mann
What is their birthday?
Alice
Birthday database updated.
Enter a name: (blank to quit)
```

ORDERED DICTIONARIES IN PYTHON 3.7

While they're still not ordered and have no "first" key-value pair, dictionaries in Python 3.7 and later will remember the insertion order of their key-value pairs if you create a sequence value from them. For example, notice the order of items in the lists made from the eggs and ham dictionaries matches the order in which they were entered:

```
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> list(eggs)
['name', 'species', 'age']
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> list(ham)
['species', 'age', 'name']
```

The dictionaries are still unordered, as you can't access items in them using integer indexes like eggs[0] or ham[2]. You shouldn't rely on this behavior, as dictionaries in older versions of Python don't remember the insertion order of key-value pairs. For example, notice how the list doesn't match the insertion order of the dictionary's key-value pairs when I run this code in Python 3.5:

```
>>> spam = {}
>>> spam['first key'] = 'value'
>>> spam['second key'] = 'value'
>>> spam['third key'] = 'value'
>>> list(spam)
['first key', 'third key', 'second key']
```

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(). 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.

In [18]:

```
spam = {'color': 'red', 'age': 42}
```

In [19]:

```
for v in spam.values():
    print(v)
```

```
red
42
```

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:

In [20]:

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

```
color
age
```

In [21]:

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

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

When you use 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.

In [22]:

```
spam = {'color': 'red', 'age': 42}
```

In [23]:

```
spam.keys()
```

Out[23]:

```
dict_keys(['color', 'age'])
```

In [24]:

```
list(spam.keys())
```

Out[24]:

```
['color', 'age']
```

In [25]:

```
spam = {'color': 'red', 'age': 42}
```

In [26]:

```
for k, v in spam.items():
    print('Key: ' + k + ' Value: ' + str(v))
```

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

Checking Whether a Key or Value Exists in a Dictionary

In [27]:

```
spam = {'name': 'Zophie', 'age': 7}
```

In [28]:

```
'name' in spam.keys()
```

Out[28]:

```
True
```

In [29]:

```
'Zophie' in spam.values()
```

Out[29]:

```
True
```

In [30]:

```
'color' in spam.keys()
```

Out[30]:

```
False
```

In [31]:

```
'color' not in spam.keys()
```

Out[31]:

```
True
```

In [32]:

```
'color' in spam
```

Out[32]:

```
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

In [33]:

```
picnicItems = {'apples': 5, 'cups': 2}
```

In [34]:

```
'I am bringing ' + str(picnicItems.get('cups', 0)) + ' cups.'
```

Out[34]:

```
'I am bringing 2 cups.'
```

In [35]:

```
'I am bringing ' + str(picnicItems.get('eggs', 0)) + ' eggs.'
```

Out[35]:

```
'I am bringing 0 eggs.'
```

In [36]:

```
picnicItems = {'apples': 5, 'cups': 2}
```

In [37]:

```
'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-37-b435afff13bf> in <module>
----> 1 'I am bringing ' + str(picnicItems['eggs']) + ' eggs.'

KeyError: 'eggs'
```

The setdefault() Method

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

In [38]:

```
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,
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.

In [39]:

```
spam = {'name': 'Pooka', 'age': 5}
```

In [40]:

```
spam.setdefault('color', 'black')
```

Out[40]:

```
'black'
```

In [41]:

```
 spam
```

Out[41]:

```
{'name': 'Pooka', 'age': 5, 'color': 'black'}
```

In [42]:

```
spam.setdefault('color', 'white')
```

Out[42]:

```
'black'
```

In [43]:

```
spam
```

Out[43]:

```
{'name': 'Pooka', 'age': 5, 'color': 'black'}
```

In [45]:

```
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)
```

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

Pretty Printing

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.

In [47]:

```
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)
```

```
{' ': 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}
```

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, you can assign each slot a string-value key, as shown in Figure.

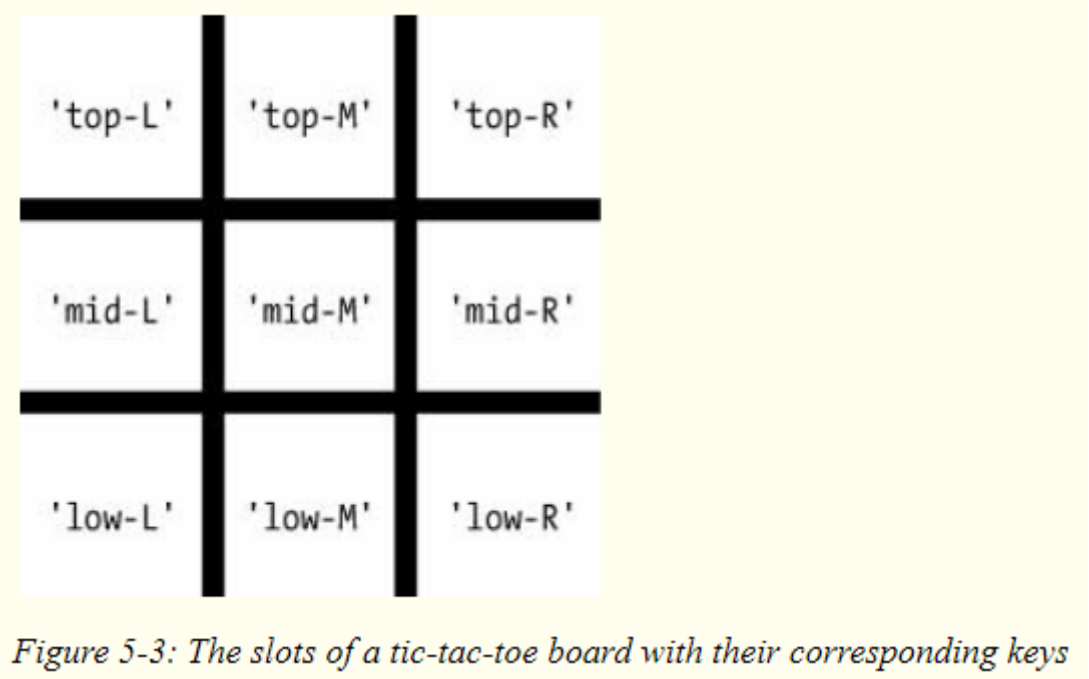


Figure 5-3: The slots of a tic-tac-toe board with their corresponding keys

In [ ]:

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

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

Nested Dictionaries and Lists

As you model more complicated things, you may find you 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.

In [ ]:

```
allGuests = {'Alice': {'apples': 5, 'pretzels': 12}, 'Bob': {'ham sandwiches': 3, 'apples':

def totalBrought(guests, item):
    numBrought = 0
    for k, v in guests.items():
        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')))
```

Summary By Quiz

MANIPULATING STRINGS

Working with Strings

String Literals

Typing string values in Python code is fairly straightforward: they begin and end with a single quote. But then how can you use a quote inside a string?
Typing 'That is Alice's cat.' won't work, because Python thinks the string ends after Alice, and the rest (s cat.') is invalid Python code. Fortunately, there are multiple ways to type strings.

Double Quotes

Strings can begin and end with double quotes, just as they do with single quotes. One benefit of using double quotes is
that the string can have a single quote character in it.

In [1]:

```
spam = "That is Alice's cat."
```

In [2]:

```
spam
```

Out[2]:

```
"That is Alice's cat."
```

Escape Characters

An escape character lets you use characters that are otherwise impossible to put into a string.
An escape character consists of a backslash () followed by the character you want to add to the string.

In [3]:

```
spam = 'Say hi to Bob\'s mother.'
```

In [4]:

```
spam
```

Out[4]:

```
"Say hi to Bob's mother."
```

Table 6-1: Escape Characters

Escape character	Prints as
\'	Single quote
\"	Double quote
\t	Tab
\n	Newline (line break)
\\	Backslash

In [5]:

```
print("Hello there!\nHow are you?\nI\'m doing fine.")
```

```
Hello there!
How are you?
I'm doing fine.
```

Raw Strings

A raw string completely ignores all escape characters and prints any backslash that appears in the string.

In [6]:

```
print(r'That is Carol\'s cat.')
```

```
That is Carol\'s cat.
```

Multiline Strings with Triple Quotes

A multiline string in Python begins and ends with either three single quotes or three double quotes. Any quotes, tabs, or newlines in between the “triple quotes” are considered part of the string.

In [7]:

```
print('''Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob''')
```

```
Dear Alice,

Eve's cat has been arrested for catnapping, cat burglary, and extortion.

Sincerely,
Bob
```

In [8]:

```
print('Dear Alice,\n\nEve\'s cat has been arrested for catnapping, cat
burglary, and extortion.\n\nSincerely,\nBob')
```

```
  File "<ipython-input-8-bd871000d7c2>", line 1
    print('Dear Alice,\n\nEve\'s cat has been arrested for catnapping, cat
                                                                          ^
SyntaxError: EOL while scanning string literal
```

Multiline Comments

While the hash character (#) marks the beginning of a comment for the rest of the line, a multiline string is often used for comments that span multiple lines.

In [10]:

```
"""This is a test Python program.
Written by Al Sweigart al@inventwithpython.com

This program was designed for Python 3, not Python 2.
"""

def spam():
    """This is a multiline comment to help
    explain what the spam() function does."""
    print('Hello!')

spam()
```

```
Hello!
```

Indexing and Slicing Strings

Strings use indexes and slices the same way lists do. You can think of the string 'Hello, world!' as a list and each character in the string as an item with a corresponding index.

In [11]:

```
spam = 'Hello, world!'
```

In [12]:

```
spam[0]
```

Out[12]:

```
'H'
```

In [13]:

```
spam[4]
```

Out[13]:

```
'o'
```

In [14]:

```
spam[-1]
```

Out[14]:

```
'!'
```

In [15]:

```
spam[0:5]
```

Out[15]:

```
'Hello'
```

In [16]:

```
spam[7:]
```

Out[16]:

```
'world!'
```

In [17]:

```
spam[:5]
```

Out[17]:

```
'Hello'
```

The in and not in Operators with Strings

The in and not in operators can be used with strings just like with list values.
An expression with two strings joined using in or not in will evaluate to a Boolean True or False.

In [18]:

```
'Hello' in 'Hello, World'
```

Out[18]:

```
True
```

In [19]:

```
'Hello' in 'Hello'
```

Out[19]:

```
True
```

In [20]:

```
'HELLO' in 'Hello, World'
```

Out[20]:

```
False
```

In [21]:

```
'' in 'spam'
```

Out[21]:

```
True
```

In [22]:

```
'cats' not in 'cats and dogs'
```

Out[22]:

```
False
```

Putting Strings Inside Other Strings

Putting strings inside other strings is a common operation in programming.
So far, we've been using the + operator and string concatenation to do this:

In [23]:
```
name = 'Al'
```

In [24]:
```
age = 4000
```

In [25]:
```
 'Hello, my name is ' + name + '. I am ' + str(age) + ' years old.'
```

Out[25]:
```
'Hello, my name is Al. I am 4000 years old.'
```

However, this requires a lot of tedious typing. A simpler approach is to use string interpolation,
in which the %s operator inside the string acts as a marker to be replaced by values following the string.
One benefit of string interpolation is that str() doesn't have to be called to convert values to strings.

In [26]:
```
name = 'Al'
```

In [27]:
```
age = 4000
```

In [28]:
```
'My name is %s. I am %s years old.' % (name, age)
```

Out[28]:
```
'My name is Al. I am 4000 years old.'
```

Python 3.6 introduced f-strings, which is similar to string interpolation except that braces are used instead of %s,
with the expressions placed directly inside the braces. Like raw strings, f-strings have an f prefix before
the starting quotation mark.

In [29]:
```
name = 'Al'
```

In [30]:
```
age = 4000
```

In [31]:

```
f'My name is {name}. Next year I will be {age + 1}.'
```

Out[31]:

```
'My name is Al. Next year I will be 4001.'
```

In [32]:

```
'My name is {name}. Next year I will be {age + 1}.'
```

Out[32]:

```
'My name is {name}. Next year I will be {age + 1}.'
```

Useful String Methods

The upper(), lower(), isupper(), and islower() Methods

The upper() and lower() string methods return a new string where all the letters in the original string have been converted to uppercase or lowercase, respectively.
Nonletter characters in the string remain unchanged.

In [33]:

```
spam = 'Hello, world!'
```

In [34]:

```
spam = spam.upper()
```

In [35]:

```
spam
```

Out[35]:

```
'HELLO, WORLD!'
```

In [36]:

```
spam = spam.lower()
```

In [37]:

```
spam
```

Out[37]:

```
'hello, world!'
```

The upper() and lower() methods are helpful if you need to make a case-insensitive comparison.

For example, the strings 'great' and 'GREat' are not equal to each other. But in the following small program, it does not matter whether the user types Great, GREAT, or grEAT, because the string is first converted to lowercase.

In [38]:

```
print('How are you?')
feeling = input()
if feeling.lower() == 'great':
    print('I feel great too.')
else:
    print('I hope the rest of your day is good.')
```

```
How are you?
great
I feel great too.
```

The isupper() and islower() methods will return a Boolean True value if the string has at least one letter and all the letters are uppercase or lowercase, respectively.
Otherwise, the method returns False.

In [39]:

```
spam = 'Hello, world!'
```

In [40]:

```
spam.islower()
```

Out[40]:

```
False
```

In [41]:

```
spam.isupper()
```

Out[41]:

```
False
```

In [42]:

```
'HELLO'.isupper()
```

Out[42]:

```
True
```

In [43]:

```
'abc12345'.islower()
```

Out[43]:

```
True
```

In [44]:

```
'12345'.islower()
```

Out[44]:

```
False
```

In [45]:

```
'12345'.isupper()
```

Out[45]:

```
False
```

Since the upper() and lower() string methods themselves return strings, you can call string methods on those returned string values as well.
Expressions that do this will look like a chain of method calls.

In [46]:

```
'Hello'.upper()
```

Out[46]:

```
'HELLO'
```

In [47]:

```
'Hello'.upper().lower()
```

Out[47]:

```
'hello'
```

In [48]:

```
'Hello'.upper().lower().upper()
```

Out[48]:

```
'HELLO'
```

In [49]:

```
 'HELLO'.lower()
```

Out[49]:

```
'hello'
```

In [50]:

```
'HELLO'.lower().islower()
```

Out[50]:

```
True
```

The isX() Methods

Along with islower() and isupper(), there are several other string methods that have names beginning with the word is.
These methods return a Boolean value that describes the nature of the string. Here are some common isX string methods:

- isalpha() Returns True if the string consists only of letters and isn't blank
- isalnum() Returns True if the string consists only of letters and numbers and is not blank
- isdecimal() Returns True if the string consists only of numeric characters and is not blank
- isspace() Returns True if the string consists only of spaces, tabs, and newlines and is not blank
- istitle() Returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters

In [51]:

```
'hello'.isalpha()
```

Out[51]:

```
True
```

In [52]:

```
'hello123'.isalpha()
```

Out[52]:

```
False
```

In [53]:

```
'hello123'.isalnum()
```

Out[53]:

```
True
```

In [54]:

```
'hello'.isalnum()
```

Out[54]:

```
True
```

In [55]:

```
'123'.isdecimal()
```

Out[55]:

```
True
```

In [56]:

```
'    '.isspace()
```

Out[56]:

```
True
```

In [57]:

```
'This Is Title Case'.istitle()
```

Out[57]:

```
True
```

In [58]:

```
'This Is Title Case 123'.istitle()
```

Out[58]:

```
True
```

In [59]:

```
 'This Is not Title Case'.istitle()
```

Out[59]:

```
False
```

In [60]:

```
'This Is NOT Title Case Either'.istitle()
```

Out[60]:

```
False
```

The isX() string methods are helpful when you need to validate user input. For example, the following program repeatedly
asks users for their age and a password until they provide valid input.

In [61]:

```
while True:
    print('Enter your age:')
    age = input()
    if age.isdecimal():
        break
    print('Please enter a number for your age.')

while True:
    print('Select a new password (letters and numbers only):')
    password = input()
    if password.isalnum():
        break
    print('Passwords can only have letters and numbers.')
```

```
Enter your age:
15
Select a new password (letters and numbers only):
123456
```

The startswith() and endswith() Methods

The startswith() and endswith() methods return True if the string value they are called on begins or ends (respectively)
with the string passed to the method; otherwise, they return False.

In [62]:

```
'Hello, world!'.startswith('Hello')
```

Out[62]:

```
True
```

In [63]:

```
'Hello, world!'.endswith('world!')
```

Out[63]:

```
True
```

In [64]:

```
 'abc123'.startswith('abcdef')
```

Out[64]:

```
False
```

In [65]:

```
'abc123'.endswith('12')
```

Out[65]:

```
False
```

In [66]:

```
'Hello, world!'.startswith('Hello, world!')
```

Out[66]:

```
True
```

In [67]:

```
 'Hello, world!'.endswith('Hello, world!')
```

Out[67]:

```
True
```

The join() and split() Methods

The join() method is useful when you have a list of strings that need to be joined together into a single string value.
The join() method is called on a string, gets passed a list of strings, and returns a string.
The returned string is the concatenation of each string in the passed-in list.

In [68]:

```
 ', '.join(['cats', 'rats', 'bats'])
```

Out[68]:

```
'cats, rats, bats'
```

In [69]:

```
' '.join(['My', 'name', 'is', 'Simon'])
```

Out[69]:

```
'My name is Simon'
```

In [70]:

```
'ABC'.join(['My', 'name', 'is', 'Simon'])
```

Out[70]:

```
'MyABCnameABCisABCSimon'
```

In [71]:

```
'My name is Simon'.split()
```

Out[71]:

```
['My', 'name', 'is', 'Simon']
```

By default, the string 'My name is Simon' is split wherever whitespace characters such as the space, tab, or newline characters are found. These whitespace characters are not included in the strings in the returned list. You can pass a delimiter string to the split() method to specify a different string to split upon.

In [72]:

```
'MyABCnameABCisABCSimon'.split('ABC')
```

Out[72]:

```
['My', 'name', 'is', 'Simon']
```

In [73]:

```
'My name is Simon'.split('m')
```

Out[73]:

```
['My na', 'e is Si', 'on']
```

A common use of split() is to split a multiline string along the newline characters.

In [75]:

```
spam = '''Dear Alice,
How have you been? I am fine.
There is a container in the fridge
that is labeled "Milk Experiment."

Please do not drink it.
Sincerely,
Bob'''
```

In [76]:

```
spam.split('\n')
```

Out[76]:

```
['Dear Alice,',
 'How have you been? I am fine.',
 'There is a container in the fridge',
 'that is labeled "Milk Experiment."',
 '',
 'Please do not drink it.',
 'Sincerely,',
 'Bob']
```

Splitting Strings with the partition() Method

The partition() string method can split a string into the text before and after a separator string.
This method searches the string it is called on for the separator string it is passed,
and returns a tuple of three substrings for the “before,” “separator,” and “after” substrings.

In [77]:

```
'Hello, world!'.partition('w')
```

Out[77]:

```
('Hello, ', 'w', 'orld!')
```

In [78]:

```
'Hello, world!'.partition('world')
```

Out[78]:

```
('Hello, ', 'world', '!')
```

If the separator string you pass to partition() occurs multiple times in the string that partition() calls on,
the method splits the string only on the first occurrence:

In [79]:

```
'Hello, world!'.partition('o')
```

Out[79]:

```
('Hell', 'o', ', world!')
```

If the separator string can’t be found, the first string returned in the tuple will be the entire string,
and the other two strings will be empty:

In [80]:

```
'Hello, world!'.partition('XYZ')
```

Out[80]:

```
('Hello, world!', '', '')
```

In [81]:

```
before, sep, after = 'Hello, world!'.partition(' ')
```

In [82]:

```
before
```

Out[82]:

```
'Hello,'
```

In [83]:

```
after
```

Out[83]:

```
'world!'
```

Justifying Text with the rjust(), ljust(), and center() Methods

The rjust() and ljust() string methods return a padded version of the string they are called on, with spaces inserted to justify the text. The first argument to both methods is an integer length for the justified string.

In [84]:

```
'Hello'.rjust(10)
```

Out[84]:

```
'     Hello'
```

In [85]:

```
 'Hello'.rjust(20)
```

Out[85]:

```
'               Hello'
```

In [86]:

```
'Hello, World'.rjust(20)
```

Out[86]:

```
'        Hello, World'
```

In [87]:

```
 'Hello'.ljust(10)
```

Out[87]:

```
'Hello     '
```

An optional second argument to rjust() and ljust() will specify a fill character other than a space character.

In [88]:

```
'Hello'.rjust(20, '*')
```

Out[88]:

```
'***************Hello'
```

In [89]:

```
'Hello'.ljust(20, '-')
```

Out[89]:

```
'Hello---------------'
```

The center() string method works like ljust() and rjust() but centers the text rather than justifying it to the left or right.

In [90]:

```
'Hello'.center(20)
```

Out[90]:

```
'       Hello        '
```

In [91]:

```
'Hello'.center(20, '=')
```

Out[91]:

```
'=======Hello========'
```

In [92]:

```
def printPicnic(itemsDict, leftWidth, rightWidth):
    print('PICNIC ITEMS'.center(leftWidth + rightWidth, '-'))
    for k, v in itemsDict.items():
        print(k.ljust(leftWidth, '.') + str(v).rjust(rightWidth))

picnicItems = {'sandwiches': 4, 'apples': 12, 'cups': 4, 'cookies': 8000}
printPicnic(picnicItems, 12, 5)
printPicnic(picnicItems, 20, 6)
```

```
---PICNIC ITEMS--
sandwiches..    4
apples......   12
cups........    4
cookies..... 8000
-------PICNIC ITEMS-------
sandwiches.....     4
apples.............    12
cups...............     4
cookies.............  8000
```

Removing Whitespace with the strip(), rstrip(), and lstrip() Methods

The strip() string method will return a new string without any whitespace characters at the beginning or end. The lstrip() and rstrip() methods will remove whitespace characters from the left and right ends, respectively.

In [93]:

```
spam = '    Hello, World    '
```

In [94]:

```
spam
```

Out[94]:

```
'    Hello, World    '
```

In [95]:

```
spam.strip()
```

Out[95]:

```
'Hello, World'
```

In [96]:

```
spam.lstrip()
```

Out[96]:

```
'Hello, World    '
```

In [97]:

```
spam.rstrip()
```

Out[97]:

```
'    Hello, World'
```

Optionally, a string argument will specify which characters on the ends should be stripped.

In [98]:

```
spam = 'SpamSpamBaconSpamEggsSpamSpam'
```

In [99]:

```
spam.strip('ampS')
```

Out[99]:

```
'BaconSpamEggs'
```

Numeric Values of Characters with the ord() and chr() Functions

Computers store information as bytes—strings of binary numbers, which means we need to be able to convert

text to numbers.
Because of this, every text character has a corresponding numeric value called a Unicode code point.
For example, the numeric code point is 65 for 'A', 52 for '4', and 33 for '!'.
You can use the ord() function to get the code point of a one-character string, and the chr() function to get the one-character string of an integer code point.

In [100]:

```
ord('A')
```

Out[100]:

```
65
```

In [101]:

```
ord('4')
```

Out[101]:

```
52
```

In [102]:

```
ord('!')
```

Out[102]:

```
33
```

In [103]:

```
chr(65)
```

Out[103]:

```
'A'
```

These functions are useful when you need to do an ordering or mathematical operation on characters:

In [104]:

```
ord('B')
```

Out[104]:

```
66
```

In [105]:

```
ord('A') < ord('B')
```

Out[105]:

```
True
```

In [106]:

```
chr(ord('A'))
```

Out[106]:

```
'A'
```

In [107]:

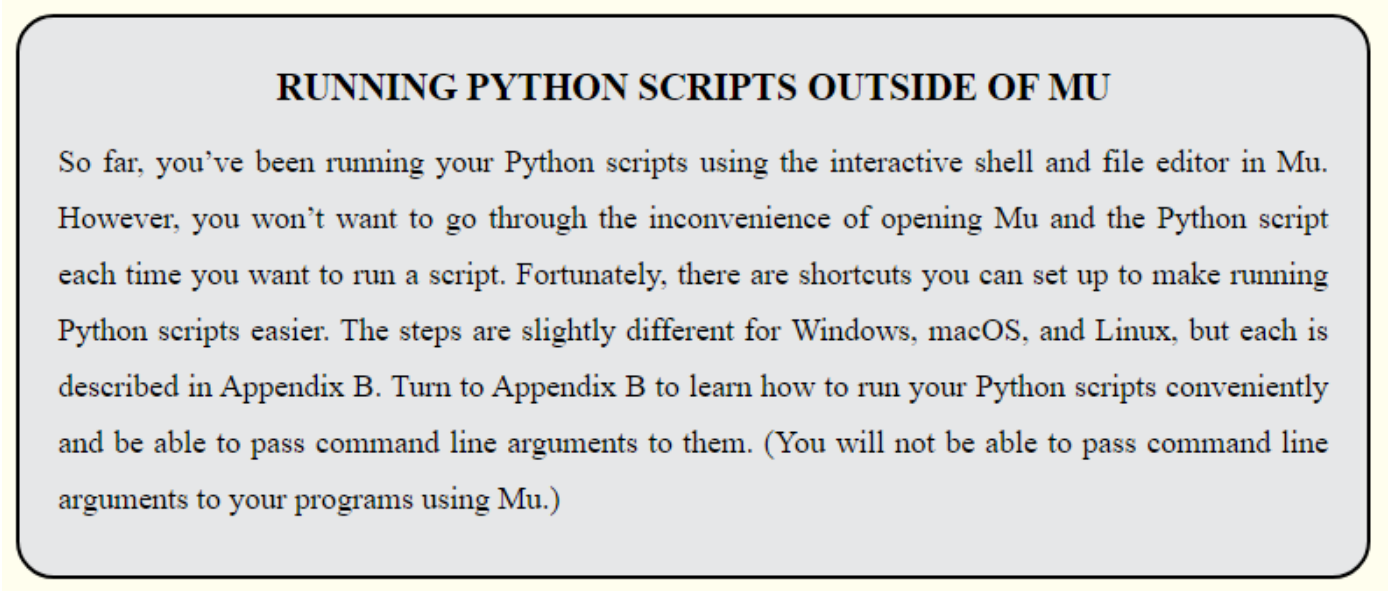
```
chr(ord('A') + 1)
```

Out[107]:

```
'B'
```

Copying and Pasting Strings with the pyperclip Module

The pyperclip module has copy() and paste() functions that can send text to and receive text from your computer's clipboard.
Sending the output of your program to the clipboard will make it easy to paste it into an email, word processor, or some other software.



RUNNING PYTHON SCRIPTS OUTSIDE OF MU

So far, you've been running your Python scripts using the interactive shell and file editor in Mu. However, you won't want to go through the inconvenience of opening Mu and the Python script each time you want to run a script. Fortunately, there are shortcuts you can set up to make running Python scripts easier. The steps are slightly different for Windows, macOS, and Linux, but each is described in Appendix B. Turn to Appendix B to learn how to run your Python scripts conveniently and be able to pass command line arguments to them. (You will not be able to pass command line arguments to your programs using Mu.)

In [109]:

```
import pyperclip
pyperclip.copy('Hello, world!')
pyperclip.paste()
```

Out[109]:

```
'Hello, world!'
```

In [110]:

```
pyperclip.paste()
```

Out[110]:

```
'pyperclip.paste()'
```

Project: Adding Bullets to Wiki Markup

The bulletPointAdder.py script will get the text from the clipboard, add a star and space to the beginning of each line,
and then paste this new text to the clipboard. For example, if I copied the following text
(for the Wikipedia article “List of Lists of Lists”) to the clipboard:

Lists of animals

Lists of aquarium life

Lists of biologists by author abbreviation

Lists of cultivars

---

and then ran the *bulletPointAdder.py* program, the clipboard would then contain the following:

---

* Lists of animals

* Lists of aquarium life

* Lists of biologists by author abbreviation

* Lists of cultivars

This star-prefixed text is ready to be pasted into a Wikipedia article as a bulleted list.

Step 1: Copy and Paste from the Clipboard

You want the bulletPointAdder.py program to do the following:

1. Paste text from the clipboard.
2. Do something to it.
3. Copy the new text to the clipboard. That second step is a little tricky, but steps 1 and 3 are pretty straightforward:
   they just involve the pyperclip.copy() and pyperclip.paste() functions. For now,
   let’s just write the part of the program that covers steps 1 and 3.

Step 2: Separate the Lines of Text and Add the Star

The call to pyperclip.paste() returns all the text on the clipboard as one big string.
If we used the “List of Lists of Lists”

Step 3: Join the Modified Lines

The lines list now contains modified lines that start with stars. But pyperclip.copy() is expecting a single string value,
however, not a list of string values. To make this single string value,
pass lines into the join() method to get a single string joined from the list’s strings.

In [111]:

```
#! python3
# bulletPointAdder.py - Adds Wikipedia bullet points to the start
# of each line of text on the clipboard.

import pyperclip
text = pyperclip.paste()

# TODO: Separate lines and add stars.

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)):    # loop through all indexes in the "lines" list
    lines[i] = '* ' + lines[i] # add star to each string in "lines" list

text = '\n'.join(lines)

pyperclip.copy(text)
```

In [112]:

```
text
```

Out[112]:

```
"* text = '\\n'.join(lines)"
```

Summary By quiz

End of Module-2

In [ ]:

```

```