Python - Lists/Array, Sets, Tuples, Dictionaries

Arrays

Array is a data-structure that can be used to store many items in one place. Imagine that we have a list of items; for example, a shopping list. We don't keep all the products on separate pages; we simply list them all together on a single page. Such a page is conceptually similar to an array. Similarly, if we plan to record air temperatures over the next 365 days, we would not create lots of individual variables, but would instead store all the data in just one array.

2.1. Creating an array

We want to create a shopping list containing three products. Such a list might be created as follows:

```
shopping = ['bread', 'butter', 'cheese']
```

(that is, shopping is the name of the array and every product within it is separated by a comma). Each item in the array is called an element. Arrays can store any number of elements (assuming that there is enough memory). Note that a list can be also empty:

shopping = []

If planning to record air temperatures over the next 365 days, we can create in advance a place to store the data. The array can be created in the following way:

```
temperatures = [0] * 365
```

(that is, we are creating an array containing 365 zeros).

2.2. Accessing array values

Arrays provide easy access to all elements. Within the array, every element is assigned a num- ber called an index. Index numbers are consecutive integers starting from 0. For example, in the array shopping = [’bread’, ’butter’, ’cheese’], ’bread’ is at index 0, ’but- ter’ is at index 1 and ’cheese’ is at index 2. If we want to check what value is located at some index (for example, at index 1), we can access it by specifing the index in square brackets, e.g. shopping[1].

2.3. Modifying array values

We can change array elements as if they were separate variables, that is each array element can be assigned a new value independently. For example, let’s say we want to record that on the 42nd day of measurement, the air temperature was 25 degrees. This can be done with a single assignment:

```
temperatures[42] = 25
```

If there was one more product to add to our shopping list, it could be appended as follows:

```
shopping += [’eggs’]
```

The index for that element will be the next integer after the last (in this case, 3).

2.4. Iterating over an array

Often we need to iterate over all the elements of an array; perhaps to count the number of specified items, for example. Knowing that the array contains of N elements, we can iterate over consecutive integers from index 0 to index N − 1 and check every such index. The length of an array can be found using the len() function. For example, counting the number of items in shopping list can be done quickly as follows:

```
N = len(shopping)
```

Let’s write a function that counts the number of days with negative air temperature.

2.1: Negative air temperature.

```
def negative(temperatures):
    N = len(temperatures)
    days=0
    for i in xrange(N):
      if temperatures[i] < 0:
      days += 1
    return days
```

Instead of iterating over indexes, we can iterate over the elements of the array.

For example, the above solution can be simplified as follows:

2.2: Negative air temperature — simplified.

```
def negative(temperatures):
    days = 0
    for t in temperatures:
      if t < 0:
      days += 0
    return days
```

2.5. Basic array operations

There are a few basic operations on arrays that are very useful. Apart from the length operation:

```
len([1, 2, 3]) == 3
```

and the repetition:

```
['Hello'] * 3 == ['Hello', 'Hello', 'Hello']
```

which we have already seen, there is also concatenation:

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

which merges two lists, and the membership operation:

```
'butter' in ['bread', 'butter', 'cheese'] == True
```

which checks for the presence of a particular item in the array.

2.6. Exercise

Problem: Given array A consisting of N integers, return the reversed array.

Solution: We can iterate over the first half of the array and exchange the elements with those in the second part of the array.

2.3 Reversing an array:

```
def reverse(A):
    N = len(A)
    for i in xrange(N//2):
      k = N - i - 1
      A[i], A[k] = A[k], A[i]
    return A
```

Python is a very rich language and provides many built-in functions and methods. It turns out, that there is already a built-in method reverse, that solves this exercise. Using such a method, array A can be reversed simply by: `A.reverse()`

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.

```
>>> supplies = ['pens', 'staplers', 'flamethrowers', 'binders']
>>> for index, item in enumerate(supplies):
...     print('Index ' + str(index) + ' in supplies is: ' + item)

#Output
#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.

```
>>> import random
>>> pets = ['Dog', 'Cat', 'Moose']
>>> random.choice(pets)
'Dog'
>>> random.choice(pets)
'Cat'
>>> random.choice(pets)
'Cat'
```

You can consider `random.choice(someList)` to be a shorter form of `someList[random.randint(0, len(someList) – 1]`.

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

```
>>> import random
>>> people = ['Alice', 'Bob', 'Carol', 'David']
>>> random.shuffle(people)
>>> people
['Carol', 'David', 'Alice', 'Bob']
>>> random.shuffle(people)
>>> people
['Alice', 'David', 'Bob', 'Carol']
```

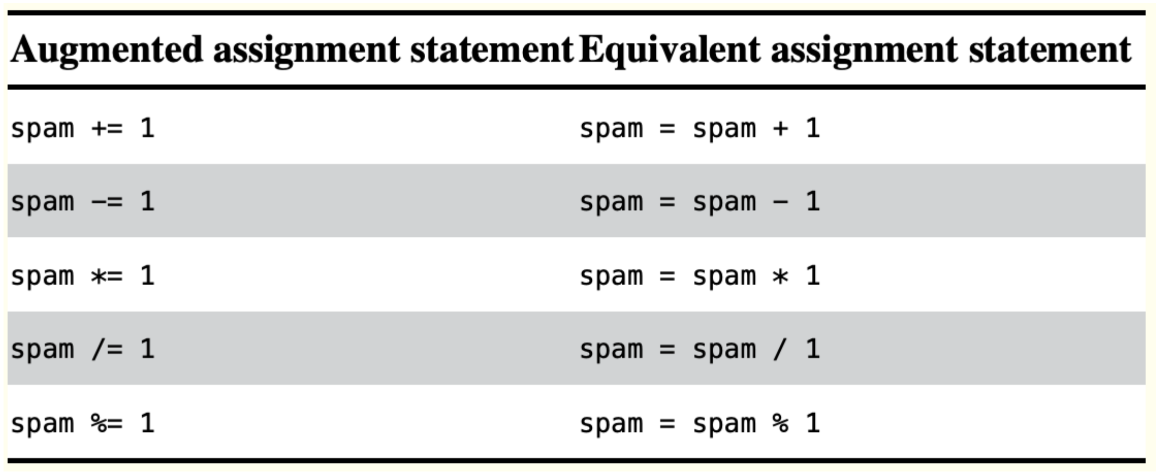
Augmented Assignment Operators

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:

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

# As a shortcut, you can use the augmented assignment operator += to do the
# same thing:
>>> spam = 42
>>> spam += 1
>>> spam
43
```

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



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`

Table 4-1 The Augmented Assignment Operators

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.

💡 The major difference between a **method** and a **function** is that a method may alter an object's state, but Python function usually only operates on it, and then prints something or returns a value.

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

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

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> spam.index('hello')
0
>>> spam.index('heyas')
3
```

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

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

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

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

The previous `append()` method call adds the argument to the end of the list. The `insert()` method can insert a value at any index in the list. The first argument to `insert()` is the index for the new value, and the second argument is the new value to be inserted.

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

💡 Notice that the code is `spam.append('moose')` and `spam.insert(1, 'chicken')`, **not** `spam = spam.append('moose')` and `spam = spam.insert(1, 'chicken')`. Neither `append()` nor `insert()` gives the new value of spam as its return value. (In fact, the return value of `append()` and `insert()` is `None`, so you definitely wouldn't want to store this as the new variable value.) Rather, the list is modified *in place.*

***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*.**

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.

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

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

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

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

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

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

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

- There are three things you should note about the `sort()` method. First, the `sort()` method sorts the list *in place*; don't try to 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.
- 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*.

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

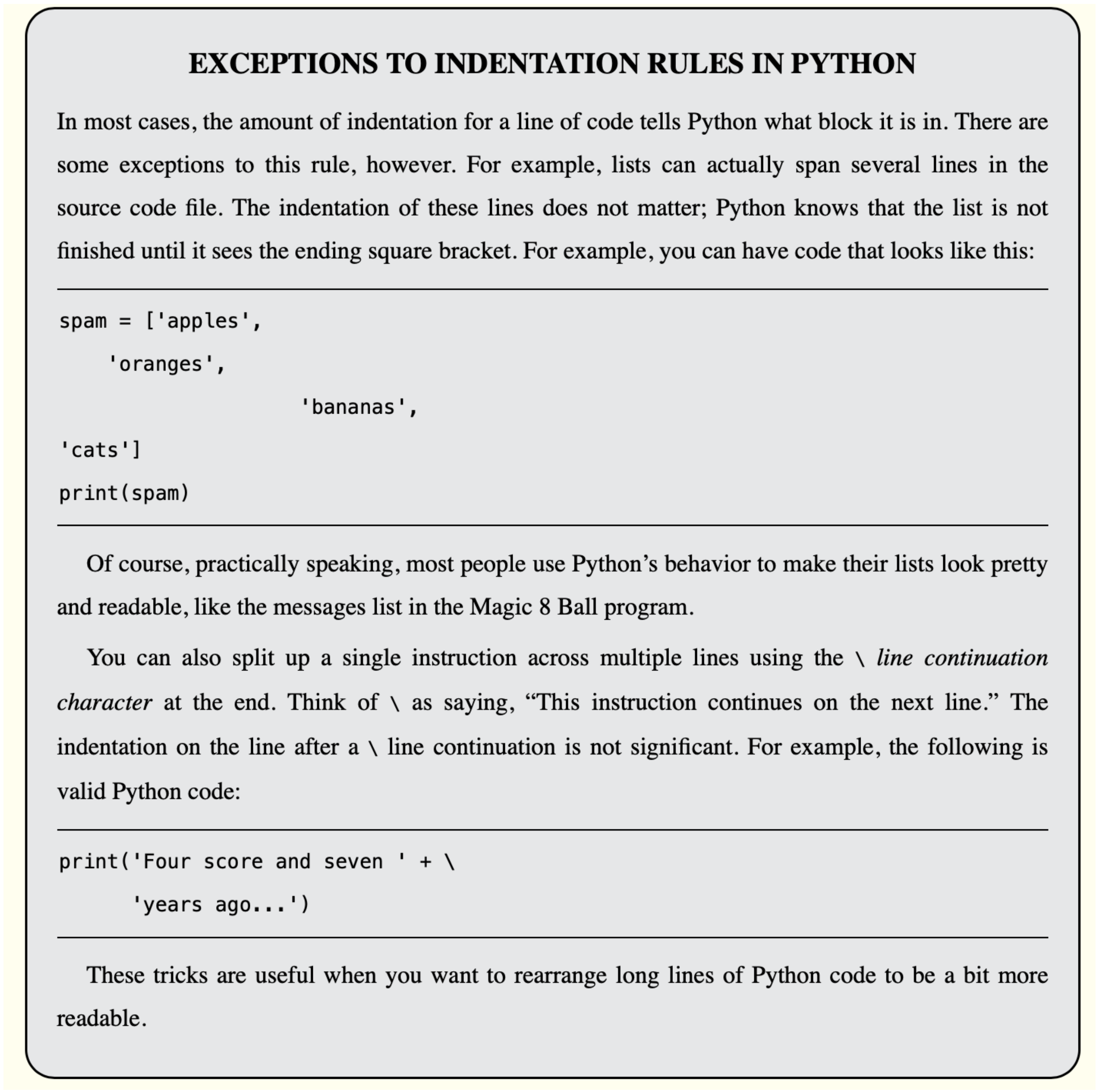
If you need to sort the values in regular alphabetical order, pass `str.lower` for the key keyword argument in the `sort()` method call.

```
>>> 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.

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.



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.

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#50>", line 1, in <module>
    name[7] = 'the'
TypeError: 'str' object does not support item assignment
```

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

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

```
>>> 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]
```

The Tuple Data Type

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

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

```
(42, 0.5)
>>> len(eggs)
3
```

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.

💡 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. (Unlike some other programming languages, it's fine to have a trailing comma after the last item in a list or tuple in Python.)

```
>>> type(('hello',))
#<class 'tuple'>
>>> type(('hello'))
#<class '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.

Identity and the id() Function

When Python runs `id('Howdy')`, it creates the `'Howdy'` string in the computer's memory. The numeric memory address where the string is stored is returned by the `id()` function. Python picks this address based on which memory bytes happen to be free on your computer at the time, so it'll be different each time you run this code.

Like all strings, `'Howdy'` is immutable and cannot be changed. If you "change" the string in a variable, a new string object is being made at a different place in memory, and the variable refers to this new string.

```
>>> bacon = 'Hello'
>>> id(bacon)
44491136
>>> bacon += ' world!' # A new string is made from 'Hello' and ' world!'.
>>> id(bacon) # bacon now refers to a completely different string.
44609712
```

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.*"**

```
>>> eggs = ['cat', 'dog'] # This creates a new list.
>>> id(eggs)
35152584
>>> eggs.append('moose') # append() modifies the list "in place".
>>> id(eggs) # eggs still refers to the same list as before.
35152584
>>> eggs = ['bat', 'rat', 'cow'] # This creates a new list, which has a new
identity.
>>> id(eggs) # eggs now refers to a completely different list.
44409800
```

If two variables refer to the same list (like spam and cheese in the previous section) and the list value itself changes, both variables are affected because they both refer to the same list. The `append()`, `extend()`, `remove()`, `sort()`, `reverse()`, and other list methods modify their lists in place.

Python's *automatic garbage collector* deletes any values not being referred to by any variables to free up memory.

A Short Program: Conway's Game of Life

Conway’s Game of Life is an example of *cellular automata*
: a set of rules governing the behavior of a field made up of discrete cells. In practice, it creates a pretty animation to look at. You can draw out each step on graph paper, using the squares as cells. A filled-in square will be “alive” and an empty square will be “dead.” If a living square has two or three living neighbors, it continues to live on the next step. If a dead square has exactly three living neighbors, it comes alive on the next step. Every other square dies or remains dead on the next step. You can see an example of the progression of steps in Figure 4-8.

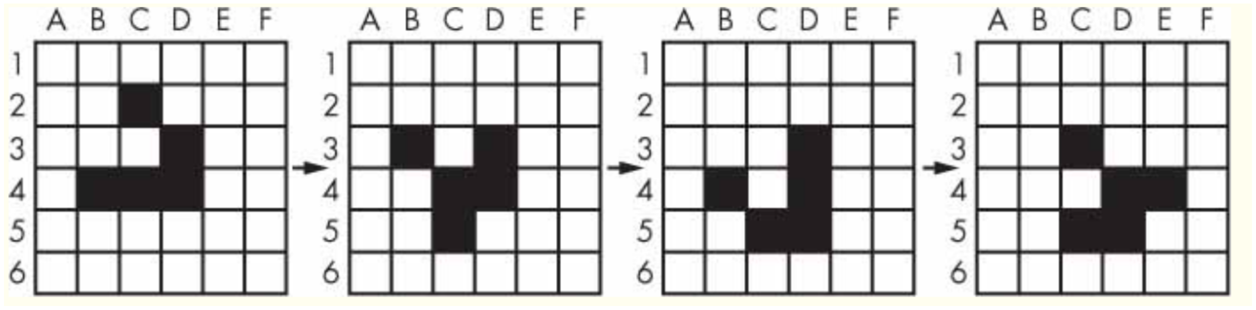


Figure 4-8: Four steps in a Conway’s Game of Life simulation

Even though the rules are simple, there are many surprising behaviors that emerge. Patterns in Conway’s Game of Life can move, self-replicate, or even mimic CPUs. But at the foundation of all of this complex, advanced behavior is a rather simple program.

We can use a list of lists to represent the two-dimensional field. The inner list represents each column of squares and stores a '#' hash string for living squares and a ' 'space string for dead squares. Type the following source code into the file editor, and save the file as *conway.py.*

Check out this video to know more about The Game of Life.

https://www.youtube.com/watch?v=jvSp6VHt_Pc

```
# 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.
```

Dictionaries and Structuring Data

The Dictionary Data Type

Like a list, 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*.

```
>>> myCat = {'size': 'fat', 'color': 'gray', 'disposition': 'loud'}
#this assigns a dictionary to the variable myCat. size, color and disposition are keys, wh
ereas fat, gray and loud are values.
```

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.

```
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.')
```