ICS3U: `join`, `split` and List Comprehensions

Merging Strings With `join`

Python contains a number of functions for processing strings. One task that we often want to perform is to take multiple values that are part of a sequence and join them together into a single string, which can be displayed, passed as an argument, or some other action.

The `join` method takes a list containing one or more string elements, and creates a new string that is composed of all of the elements in the list, joined together by one or more characters in a given string. It has the general format `"string".join(list)`. It is important to remember that `join` is a string method, even though it uses elements from a list. To join together the strings in a list, a call to `join` would look something like this.

```
>>> fruit = ["apple", "pear", "lemon", "banana"]
>>> ", ".join(fruit)
'apple, pear, lemon, banana'
```

A common mistake is to try to write it like this.

```
>>> fruit.join(", ")
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
AttributeError: 'list' object has no attribute 'join'
```

Only strings can be used with `join`. If a list contains any non-string values, then a run-time error will occur.

```
>>> integers = [3, 1, 5, 2]
>>> "-".join(integers)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: sequence item 0: expected str instance, int found
```

Breaking Strings Apart With `split`

In many ways, `split` is the "opposite" of `join`. It takes a string that contains one or more separator character and returns a list of strings split at those characters. The following code breaks apart a sequence of letters, each separated by a semicolon. The `split` method creates a list and adds each letter as a separate element.

```
>>> letters = "a;b;c;d"
>>> letters.split(";")
['a', 'b', 'c', 'd']
```

This is useful for having the user input a series of values without using a loop. Consider the code below.

```
name_list = []
print("Enter some names (enter no value to quit): ")
while True:
   name = input("Name: ")
   if name == "":
      break
   else:
      name_list.append(name)
print("The list of names is", name_list)
```

It performs essentially the same function as the following snippet, assuming the user does as instructed.

```
name_list = input("Enter some names, separated by commas: ").split(",")
print("The list of names is", name_list)
```

If no argument is provided to `split`, it uses a space for the separator character. Extra spaces are ignored.

```
>>> integers = "1    3              5 7     ".split()
>>> integers
['1', '3', '5', '7']
```

Since `split` creates a list of strings, a common mistake for new programmers is to try to read in a sequence of integers, like in the last example. If we attempt to use the values as numeric data types, we can run into trouble.

```
>>> sum(integers)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

If the values must be numeric ones, we will have to convert them using some method.

List Comprehensions

A list comprehension creates a list from a sequence, using some set of rules. Most list comprehensions have a format similar to `[expression for element in sequence]`. Any sequence type can be used in a list comprehension, including lists, tuples, strings, `range` objects, and so on. For example, the code below doubles all values in a list.

```
>>> L = [1, 2, 3]
>>> [2*x for x in L]
[2, 4, 6]
```

The expression, `2*x`, doubles each integer, `x`, in the list, `L`. The result is a new list consisting of the doubled values.

The code below iterates over all of the characters in a string and creates a list of capitalized letters.

```
>>> [s.upper() for s in "abcde"]
['A', 'B', 'C', 'D', 'E']
```

The expression, `s.upper()`, converts each character, `s`, in the string `"abcde"`. Again, the result is a list.

List comprehensions provide a handy way to convert multiple values from one data type to another at once. The following code reads some space-separated integers from the user and determines their average.

```
values = input("Enter some positive integers, separated by spaces: ").split()
values = [int(x) for x in values]
average = sum(values) / len(values)
print("The average of the integers is", average)
```

The expression, `int(x)`, converts each element, `x`, in the list, `values`, from a string to an integer. It overwrites `values` with a new list. Try running the code to verify that it performs as intended.

Note that if any value in the list *cannot* be converted to an integer, then the program will crash. We can get around this by specifying one or more conditions, following the list. List comprehensions with conditions often have the form `[expression for element in sequence if conditions]`. The code below repeats the action above, but only adds positive integers (those that are composed solely of the characters 0-9) to the resulting list. All other values will be ignored.

```
values = input("Enter some positive integers, separated by spaces: ").split()
values = [int(x) for x in values if x.isdigit()]
average = sum(values) / len(values)
print("The average of the integers is", average)
```

Sometimes conditions can appear in the expression itself. The following list comprehension takes the values 0-9 and creates a list consisting of “Even” or “Odd” for each value.

```
>>> ["Even" if x % 2 == 0 else "Odd" for x in range(10)]
['Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd']
```

Since `x % 2 == 0` is `True` when `x` is 0, the first entry in the list is `Even`. Since `x % 2 == 0` is `False` when `x` is 1, the second entry in the list is `Odd`.

List comprehensions can get fairly complex, and may include nested structures like `if` blocks or loops. In promotion of readability, it may be better to simply avoid the use of list comprehensions in these cases.