Strings and Lists

Two of the most useful methods on strings involve lists of strings. The `split` method breaks a string into a list of words. By default, any number of whitespace characters is considered a word boundary.

```
song = "The rain in Spain..."
wds = song.split()
print(wds)

```

ActiveCode: 1 (ch09_split1)

Run

```
['The', 'rain', 'in', 'Spain...']
```

An optional argument called a **delimiter** can be used to specify which characters to use as word boundaries. The following example uses the string `ai` as the delimiter:

```
song = "The rain in Spain..."
wds = song.split('ai')
print(wds)

```

ActiveCode: 2 (ch09_split2)

Run

```
['The r', 'n in Sp', 'n...']
```

Notice that the delimiter doesn't appear in the result.

The inverse of the `split` method is `join`. You choose a desired **separator** string, (often called the *glue*) and join the list with the glue between each of the elements.

```
wds = ["red", "blue", "green"]
glue = ';'
s = glue.join(wds)
print(s)
print(wds)

print("***".join(wds))
print("".join(wds))

```

ActiveCode: 3 (ch09_join)

Run

```
red;blue;green
['red', 'blue', 'green']
red***blue***green
redbluegreen
```

The list that you glue together (`wds` in this example) is not modified. Also, you can use empty glue or multi-character strings as glue.

Check your understanding

`list` Type Conversion Function

Python has a built-in type conversion function called `list` that tries to turn whatever you give it into a list. For example, try the following:

```
xs = list("Crunchy Frog")
print(xs)

```

ActiveCode: 4 (ch09_list1)

Run

```
['C', 'r', 'u', 'n', 'c', 'h', 'y', ' ', 'F', 'r', 'o', 'g']
```

The string “Crunchy Frog” is turned into a list by taking each character in the string and placing it in a list. In general, any sequence can be turned into a list using this function. The result will be a list containing the elements in the original sequence. It is not legal to use the `list` conversion function on any argument that is not a sequence.

It is also important to point out that the `list` conversion function will place each element of the original sequence in the new list. When working with strings, this is very different than the result of the `split` method. Whereas `split` will break a string into a list of “words”, `list` will always break it into a list of characters.

Tuples and Mutability

So far you have seen two types of sequential collections: strings, which are made up of characters; and lists, which are made up of elements of any type. One of the differences we noted is that the elements of a list can be modified, but the characters in a string cannot. In other words, strings are **immutable** and lists are **mutable**.

A **tuple**, like a list, is a sequence of items of any type. Unlike lists, however, tuples are immutable. Syntactically, a tuple is a comma-separated sequence of values. Although it is not necessary, it is conventional to enclose tuples in parentheses:

```
julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia")
```

Tuples are useful for representing what other languages often call *records* — some related information that belongs together, like your student record. There is no description of what each of these *fields* means, but we can guess. A tuple lets us “chunk” together related information and use it as a single thing.

Tuples support the same sequence operations as strings and lists. For example, the index operator selects an element from a tuple.

As with strings, if we try to use item assignment to modify one of the elements of the tuple, we get an error.

```
julia[0] = 'X'
TypeError: 'tuple' object does not support item assignment
```

Of course, even if we can’t modify the elements of a tuple, we can make a variable reference a new tuple holding different information. To construct the new tuple, it is convenient that we can slice parts of the old tuple and join up the bits to make the new tuple. So `julia` has a new recent film, and we might want to change her tuple. We can easily slice off the parts we want and concatenate them with the new tuple.

```
julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta, Georgia")
print(julia[2])
print(julia[2:6])

print(len(julia))

julia = julia[:3] + ("Eat Pray Love", 2010) + julia[5:]
print(julia)

```

ActiveCode: 5 (ch09_tuple1)

Run

```
1967
1967 Duplicity 2009 Actress
7
Julia Roberts 1967 Eat Pray Love 2010 Actress Atlanta, Georgia
```

To create a tuple with a single element (but you're probably not likely to do that too often), we have to include the final comma, because without the final comma, Python treats the `(5)` below as an integer in parentheses:

```
tup = (5,)
print(type(tup))

x = (5)
print(type(x))

```

ActiveCode: 6 (chp09_tuple2)

Run

```
<type 'tuple'>
<type 'int'>
```

Tuple Assignment

Python has a very powerful **tuple assignment** feature that allows a tuple of variables on the left of an assignment to be assigned values from a tuple on the right of the assignment.

```
(name, surname, birth_year, movie, movie_year, profession, birth_place) = julia
```

This does the equivalent of seven assignment statements, all on one easy line. One requirement is that the number of variables on the left must match the number of elements in the tuple.

Once in a while, it is useful to swap the values of two variables. With conventional assignment statements, we have to use a temporary variable. For example, to swap `a` and `b`:

```
temp = a
a = b
b = temp
```

Tuple assignment solves this problem neatly:

```
(a, b) = (b, a)
```

The left side is a tuple of variables; the right side is a tuple of values. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. This feature makes tuple assignment quite versatile.

Naturally, the number of variables on the left and the number of values on the right have to be the same.

```
>>> (a, b, c, d) = (1, 2, 3)
ValueError: need more than 3 values to unpack
```

Tuples as Return Values

Functions can return tuples as return values. This is very useful — we often want to know some batsman's highest and lowest score, or we want to find the mean and the standard deviation, or we want to know the year, the month, and the day, or if we're doing some some ecological modeling we may want to know the number of rabbits and the number of wolves on an island at a given time. In each case, a function (which can only return a single value), can create a single tuple holding multiple elements.

For example, we could write a function that returns both the area and the circumference of a circle of radius r.

```
def circleInfo(r):
    """ Return (circumference, area) of a circle of radius r """
    c = 2 * 3.14159 * r
    a = 3.14159 * r * r
    return (c, a)

print(circleInfo(10))

```

ActiveCode: 7 (chp09_tuple3)

Run

```
62.8318 314.159
```

Note

This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

```

```

ActiveCode: 8 (scratch_09_07)

Run