# 9. Tuples

## 9.1. Tuples are used for grouping data

We saw earlier that we could group together pairs of values by surrounding with parentheses. Recall this example:

```
>>> year_born = ("Paris Hilton", 1981)
```

This is an example of a **data structure** — a mechanism for grouping and organizing data to make it easier to use.

The pair is an example of a **tuple**. Generalizing this, a tuple can be used to group any number of items into a single compound value. 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, Geo
```

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. The index operator selects an element from a tuple.

```
>>> julia[2]
1967
```

But 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
```

So like strings, tuples are immutable. Once Python has created a tuple in memory, it cannot be changed.

Of course, even if we can't modify the elements of a tuple, we can always make the `julia` 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 if `julia` has a new recent film, we could change her variable to reference a new tuple that used some information from the old one:

```
>>> julia = julia[:3] + ("Eat Pray Love", 2010) + julia[5:]
>>> julia
("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,)
>>> type(tup)
<class 'tuple'>
>>> x = (5)
>>> type(x)
<class 'int'>
```

## 9.2. 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. (We already saw this used for pairs, but it generalizes.)

```
(name, surname, b_year, movie, m_year, profession, b_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.

One way to think of tuple assignment is as tuple packing/unpacking.

In tuple packing, the values on the left are 'packed' together in a tuple:

```
>>> b = ("Bob", 19, "CS")    # tuple packing
```

In tuple unpacking, the values in a tuple on the right are 'unpacked' into the variables/names on the right:

```
>>> b = ("Bob", 19, "CS")
>>> (name, age, studies) = b    # tuple unpacking
>>> name
'Bob'
>>> age
19
>>> studies
'CS'
```

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
```

## 9.3. Tuples as return values

Functions can always only return a single value, but by making that value a tuple, we can effectively group together as many values as we like, and return them together. 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 modelling we may want to know the number of rabbits and the number of wolves on an island at a given time.

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

```
def f(r):
    """ Return (circumference, area) of a circle of radius r """
    c = 2 * math.pi * r
    a = math.pi * r * r
    return (c, a)
```

## 9.4. Composability of Data Structures

We saw in an earlier chapter that we could make a list of pairs, and we had an example where one of the items in the tuple was itself a list:

```
students = [
    ("John", ["CompSci", "Physics"]),
    ("Vusi", ["Maths", "CompSci", "Stats"]),
    ("Jess", ["CompSci", "Accounting", "Economics", "Management"]),
    ("Sarah", ["InfSys", "Accounting", "Economics", "CommLaw"]),
    ("Zuki", ["Sociology", "Economics", "Law", "Stats", "Music"])]
```

Tuples items can themselves be other tuples. For example, we could improve the information about our movie stars to hold the full date of birth rather than just the year, and we could have a list of some of her movies and dates that they were made, and so on:

```
julia_more_info = ( ("Julia", "Roberts"), (8, "October", 1967),
                    "Actress", ("Atlanta", "Georgia"),
                    [ ("Duplicity", 2009),
                      ("Notting Hill", 1999),
                      ("Pretty Woman", 1990),
                      ("Erin Brockovich", 2000),
                      ("Eat Pray Love", 2010),
                      ("Mona Lisa Smile", 2003),
                      ("Oceans Twelve", 2004) ])
```

Notice in this case that the tuple has just five elements — but each of those in turn can be another tuple, a list, a string, or any other kind of Python value. This property is known as being **heterogeneous**, meaning that it can be composed of elements of different types.

## 9.5. Glossary

**data structure**

An organization of data for the purpose of making it easier to use.

**immutable data value**

A data value which cannot be modified. Assignments to elements or slices (sub-parts) of immutable values cause a runtime error.

**mutable data value**

A data value which can be modified. The types of all mutable values are compound types. Lists and dictionaries are mutable; strings and tuples are not.

**tuple**

An immutable data value that contains related elements. Tuples are used to group together related data, such as a person's name, their age, and their gender.

**tuple assignment**

An assignment to all of the elements in a tuple using a single assignment statement. Tuple assignment occurs *simultaneously* rather than in sequence, making it useful for swapping values.

## 9.6. Exercises

1. We've said nothing in this chapter about whether you can pass tuples as arguments to a function. Construct a small Python example to test whether this is possible, and write up your findings.
2. Is a pair a generalization of a tuple, or is a tuple a generalization of a pair?
3. Is a pair a kind of tuple, or is a tuple a kind of pair?