6.4 Tuples

Old programming languages such as C have strict rules for the programmer for the ways in which data must be handled. The burden of showing that a program is consistent in its data usage is put on the programmer. As a result, programs in these languages can run very efficiently; little needs to be checked during the execution of a program. Python reverses this. It places little burden on the programmer, and this means that things that are verified in other languages when the program is written need to be checked during run-time on Python programs. For the most part this is a good trade-off; it enables Python programs to be written by many people who are not professional programmers, and processors have become so fast that we seldom notice the increased time for the execution. Occasionally, however, this tradeoff does not work well for us; we would like to get faster execution, even at the cost of more elaboration at programming time. One way to achieve this is to use data structures with less flexibility.

We have seen that the list data type in Python is useful in many situations because of its mutability. We can create an empty list and gradually add members to it as we have a need to do so. We can insert members, delete members and alter members of a list whenever we need to. This gives great power to the programmer, but it also requires that every list be created with the ability to expand without limits. Python has a similar data type called a *tuple* that has many of the same properties as lists, but tuples are immutable. Once a tuple is created, its contents cannot be altered. Tuples place fewer demands on the system, so programs that use them might run faster than similar programs that use lists. More important for our purposes, there are things that Python will let you do with tuples that you cannot do with lists. Among the most important of these is serving as keys for dictionaries. Tuples can be keys, lists cannot.

Tuples are written as comma-separated sequences of values inside parentheses, as in `(3, 4, 5)`. The parentheses can be eliminated if you choose: the expression `2, 3` is interpreted by Python as the tuple whose first element is 2 and whose second is 3; this is more commonly written `(2, 3)`. If you need a tuple with only one element 2, it is written `(2,)`. Tuples are made differently than lists: we often start with an empty list and build it up one element at a time, but since tuples are immutable we have to start tuples with their complete values.

Tuples are indexed the same way as lists and dictionaries, with square brackets. If `T` is a tuple, then `T[0]` is its first element, `T[1]` is the second element, and so forth.

One thing that you can do in Python that is not allowed in older languages is assigning to a structure in which the entries are variables. For example, Python allows the statement

```
(x, y) = (3, 5)
```

and the effect of this statement is to assign 3 to `x` and 5 to `y`. This works with both the tuple `(x, y)` and the list `[x, y]`, but since we would be unlikely to alter

a structure containing several variables, tuples are the more natural structure to use for such an assignment.

The following program illustrates the way we could use this to have a function return multiple values. Here function MinMax( ) takes as an argument a list of numeric values and returns a tuple containing the smallest and largest values in the list. Note that in main( ) we can assign a tuple of variables to the value returned by a call to MinMax( )

```
# This finds the largest and smallest values of a list.
def MinMax(L):
    # This returns the largest and smallest values of list L.
    # We assume that all the values in L are between 0
    # and 100.
    min = 101
    max = -1
    for x in L:
        if x < min:
            min = x
        if x > max:
            max = x
    return (min, max)

def main():
    L = [2, 9, 1, 3, 5, 4, 2, 7, 5]
    (m, M) = MinMax(L)
    print( "The smallest of those values is %d" % m )
    print( "The largest of those values is %d" % M )

main( )
```

Program 6.4.1: Largest and smallest values of a list

. In the next program we store and edit a list of names, where names are represented by (givenNames, familyName) tuples. Since we do not plan to edit the names after they are created, tuples are the natural structure here. Since tuples, like strings, are immutable, we could use such a structure any place where we use strings, such as for the keys of a dictionary.

Our program needs to take a number of possibilities into account. Some people have only one name, as in "Madonna" or "Prince". We will ultimately sort the names by the family name field, so we will put this single name as the family name, which is the second entry in the tuple, and for these use an empty string in the given name field. Some people have more than one given name, as in "Barack Hussein Obama" or "Charles Philip Arthur George Windsor". We

will assume that the last of the words in such a string is the family name and put everything else into one long given name.

```
# This reads a list of names and sorts it.
# The names are stored as (givenName, familyName)
def ReadName():
    response = input( "Enter a name: " )
    names = response.split()
    if len(names) == 0:
        return ("", "")
    else:
        lastName = names[ len(names) - 1 ]
        if len(names) == 1:
            firstName = ""
        else:
            firstName = names[ 0 ]
        return (firstName, lastName )

def PrintList(L):
    # This prints list L
    for entry in L:
        print( "%s %s" % (entry[0], entry[1]) )

def compare(name1, name2):
    # This returns -1 if name1 < name2,
    # 0 if name1 == name2,
    # and 1 if name1 > name2
    if name1[1] < name2[1]:
        return -1
    elif name1[1] > name2[1]:
        return 1
    elif name1[0] < name2[0]:
        return -1
    elif name1[0] > name2[0]:
        return 1
    else:
        return 0

def main():
    PersonList = []
    done = False
    while not done:
        p = ReadName()
        if p[1] == "":
            done = True
        else:
            PersonList.append(p)
    PersonList.sort (compare)  # Note this use of compare
    PrintList( PersonList )

main()
```

Program 6.4.2: Sorting (givenName, familyName) tuples