10: TUPLES

Chuck Severance
University of Michigan

LibreTexts™

CHAPTER OVERVIEW

10: TUPLES

10.1: TUPLES ARE IMMUTABLE

A tuple1 is a sequence of values much like a list. The values stored in a tuple can be any type, and they are indexed by integers. The important difference is that tuples are immutable. Tuples are also comparable and hashable so we can sort lists of them and use tuples as key values in Python dictionaries.

10.2: COMPARING TUPLES

The comparison operators work with tuples and other sequences. Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next element, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

10.3: TUPLE ASSIGNMENT

One of the unique syntactic features of the Python language is the ability to have a tuple on the left side of an assignment statement. This allows you to assign more than one variable at a time when the left side is a sequence.

10.4: DICTIONARIES AND TUPLES

Dictionaries have a method called items that returns a list of tuples, where each tuple is a key-value pair.

10.5: MULTIPLE ASSIGNMENT WITH DICTIONARIES

10.6: THE MOST COMMON WORDS

10.7: USING TUPLES AS KEYS IN DICTIONARIES

Because tuples are hashable and lists are not, if we want to create a composite key to use in a dictionary we must use a tuple as the key.

10.8: SEQUENCES- STRINGS, LISTS, AND TUPLES - OH MY!

10.9: DEBUGGING

in this chapter we are starting to see compound data structures, like lists of tuples, and dictionaries that contain tuples as keys and lists as values. Compound data structures are useful, but they are prone to what I call shape errors; that is, errors caused when a data structure has the wrong type, size, or composition, or perhaps you write some code and forget the shape of your data and introduce an error.

10.E: TUPLES (EXERCISES)

10.G: TUPLES (GLOSSARY)

10.1: Tuples are Immutable

A tuple[1] is a sequence of values much like a list. The values stored in a tuple can be any type, and they are indexed by integers. The important difference is that tuples are *immutable*. Tuples are also *comparable* and *hashable* so we can sort lists of them and use tuples as key values in Python dictionaries.

Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses to help us quickly identify tuples when we look at Python code:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, you have to include the final comma:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

Without the comma Python treats `('a')` as an expression with a string in parentheses that evaluates to a string:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

Another way to construct a tuple is the built-in function `tuple`. With no argument, it creates an empty tuple:

```
>>> t = tuple()
>>> print(t)
()
```

If the argument is a sequence (string, list, or tuple), the result of the call to `tuple` is a tuple with the elements of the sequence:

```
>>> t = tuple('lupins')
>>> print(t)
('l', 'u', 'p', 'i', 'n', 's')
```

Because `tuple` is the name of a constructor, you should avoid using it as a variable name.

Most list operators also work on tuples. The bracket operator indexes an element:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

And the slice operator selects a range of elements.

```
>>> print(t[1:3])
('b', 'c')
```

But if you try to modify one of the elements of the tuple, you get an error:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

You can't modify the elements of a tuple, but you can replace one tuple with another:

```
>>> t = ('A',) + t[1:]
>>> print(t)
('A', 'b', 'c', 'd', 'e')
```

10.2: Comparing Tuples

The comparison operators work with tuples and other sequences. Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next element, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

The `sort` function works the same way. It sorts primarily by first element, but in the case of a tie, it sorts by second element, and so on.
This feature lends itself to a pattern called *DSU* for

Decorate

a sequence by building a list of tuples with one or more sort keys preceding the elements from the sequence,

Sort

the list of tuples using the Python built-in `sort`, and

Undecorate

by extracting the sorted elements of the sequence.

[DSU]
For example, suppose you have a list of words and you want to sort them from longest to shortest:

```
%%python3

txt = 'but soft what light in yonder window breaks'
words = txt.split()
t = list()
for word in words:
    t.append((len(word), word))

t.sort(reverse=True)

res = list()
for length, word in t:
    res.append(word)

print(res)

# Code: http://www.py4e.com/code3/soft.py
```

run | restart | restart & run all

The first loop builds a list of tuples, where each tuple is a word preceded by its length.
`sort` compares the first element, length, first, and only considers the second element to break ties. The keyword argument `reverse=True` tells `sort` to go in decreasing order.
The second loop traverses the list of tuples and builds a list of words in descending order of length. The four-character words are sorted in *reverse* alphabetical order, so "what" appears before "soft" in the following list.

The output of the program is as follows:

```
['yonder', 'window', 'breaks', 'light', 'what',
'soft', 'but', 'in']
```

Of course the line loses much of its poetic impact when turned into a Python list and sorted in descending word length order.

10.3: Tuple Assignment

One of the unique syntactic features of the Python language is the ability to have a tuple on the left side of an assignment statement. This allows you to assign more than one variable at a time when the left side is a sequence.

In this example we have a two-element list (which is a sequence) and assign the first and second elements of the sequence to the variables `x` and `y` in a single statement.

```
>>> m = [ 'have', 'fun' ]
>>> x, y = m
>>> x
'have'
>>> y
'fun'
>>>
```

It is not magic, Python *roughly* translates the tuple assignment syntax to be the following:[2]

```
>>> m = [ 'have', 'fun' ]
>>> x = m[0]
>>> y = m[1]
>>> x
'have'
>>> y
'fun'
>>>
```

Stylistically when we use a tuple on the left side of the assignment statement, we omit the parentheses, but the following is an equally valid syntax:

```
>>> m = [ 'have', 'fun' ]
>>> (x, y) = m
>>> x
'have'
>>> y
'fun'
>>>
```

A particularly clever application of tuple assignment allows us to *swap* the values of two variables in a single statement:

```
>>> a, b = b, a
```

Both sides of this statement are tuples, but the left side is a tuple of variables; the right side is a tuple of expressions. Each value on the right side is assigned to its respective variable on the left side. All the expressions on the right side are evaluated before any of the assignments.

The number of variables on the left and the number of values on the right must be the same:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

More generally, the right side can be any kind of sequence (string, list, or tuple). For example, to split an email address into a user name and a domain, you could write:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

The return value from `split` is a list with two elements; the first element is assigned to `uname`, the second to `domain`.

```
>>> print(uname)
monty
>>> print(domain)
python.org
```

10.4: Dictionaries and Tuples

Dictionaries have a method called `items` that returns a list of tuples, where each tuple is a key-value pair:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> print(t)
[('b', 1), ('a', 10), ('c', 22)]
```

As you should expect from a dictionary, the items are in no particular order.

However, since the list of tuples is a list, and tuples are comparable, we can now sort the list of tuples. Converting a dictionary to a list of tuples is a way for us to output the contents of a dictionary sorted by key:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> t
[('b', 1), ('a', 10), ('c', 22)]
>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

The new list is sorted in ascending alphabetical order by the key value.

10.5: Multiple assignment with dictionaries

Combining `items` , tuple assignment, and `for` , you can see a nice code pattern for traversing the keys and values of a dictionary in a single loop:

```
%%python3
d = {'a':10, 'b':1, 'c':22}

for key, val in list(d.items()):
    print(val, key)
```

run | restart | restart & run all

This loop has two *iteration variables* because `items` returns a list of tuples and `key, val` is a tuple assignment that successively iterates through each of the key-value pairs in the dictionary.
For each iteration through the loop, both `key` and `value` are advanced to the next key-value pair in the dictionary (still in hash order).
The output of this loop is:

```
10 a
22 c
1 b
```

Again, it is in hash key order (i.e., no particular order).
If we combine these two techniques, we can print out the contents of a dictionary sorted by the *value* stored in each key-value pair.
To do this, we first make a list of tuples where each tuple is `(value, key)` . The `items` method would give us a list of `(key, value)` tuples, but this time we want to sort by value, not key. Once we have constructed the list with the value-key tuples, it is a simple matter to sort the list in reverse order and print out the new, sorted list.

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for key, val in d.items() :
...     l.append( (val, key) )
...
>>> l
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
>>>
```

By carefully constructing the list of tuples to have the value as the first element of each tuple, we can sort the list of tuples and get our dictionary contents sorted by value.

10.6: The most common words

Coming back to our running example of the text from *Romeo and Juliet* Act 2, Scene 2, we can augment our program to use this technique to print the ten most common words in the text as follows:

```
import string
fhand = open('romeo-full.txt')
counts = dict()
for line in fhand:
    line = line.translate(str.maketrans('', '', string.punctuation))
    line = line.lower()
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1

# Sort the dictionary by value
lst = list()
for key, val in list(counts.items()):
    lst.append((val, key))

lst.sort(reverse=True)

for key, val in lst[:10]:
    print(key, val)

# Code: http://www.py4e.com/code3/count3.py
```

The first part of the program which reads the file and computes the dictionary that maps each word to the count of words in the document is unchanged. But instead of simply printing out `counts` and ending the program, we construct a list of `(val, key)` tuples and then sort the list in reverse order.

Since the value is first, it will be used for the comparisons. If there is more than one tuple with the same value, it will look at the second element (the key), so tuples where the value is the same will be further sorted by the alphabetical order of the key.

At the end we write a nice `for` loop which does a multiple assignment iteration and prints out the ten most common words by iterating through a slice of the list (`lst[:10]`).

So now the output finally looks like what we want for our word frequency analysis.

```
61 i
42 and
40 romeo
34 to
34 the
32 thou
32 juliet
30 that
29 my
24 thee
```

The fact that this complex data parsing and analysis can be done with an easy-to-understand 19-line Python program is one reason why Python is a good choice as a language for exploring information.

10.7: Using Tuples as Keys in Dictionaries

Because tuples are *hashable* and lists are not, if we want to create a *composite* key to use in a dictionary we must use a tuple as the key.

We would encounter a composite key if we wanted to create a telephone directory that maps from last-name, first-name pairs to telephone numbers. Assuming that we have defined the variables `last`, `first`, and `number`, we could write a dictionary assignment statement as follows:

```
directory[last,first] = number
```

The expression in brackets is a tuple. We could use tuple assignment in a `for` loop to traverse this dictionary.

```
for last, first in directory:
    print(first, last, directory[last,first])
```

This loop traverses the keys in `directory`, which are tuples. It assigns the elements of each tuple to `last` and `first`, then prints the name and corresponding telephone number.

10.8: Sequences- strings, lists, and tuples - Oh My!

I have focused on lists of tuples, but almost all of the examples in this chapter also work with lists of lists, tuples of tuples, and tuples of lists. To avoid enumerating the possible combinations, it is sometimes easier to talk about sequences of sequences.

In many contexts, the different kinds of sequences (strings, lists, and tuples) can be used interchangeably. So how and why do you choose one over the others?

To start with the obvious, strings are more limited than other sequences because the elements have to be characters. They are also immutable. If you need the ability to change the characters in a string (as opposed to creating a new string), you might want to use a list of characters instead.

Lists are more common than tuples, mostly because they are mutable. But there are a few cases where you might prefer tuples:

1. In some contexts, like a `return` statement, it is syntactically simpler to create a tuple than a list. In other contexts, you might prefer a list.
2. If you want to use a sequence as a dictionary key, you have to use an immutable type like a tuple or string.
3. If you are passing a sequence as an argument to a function, using tuples reduces the potential for unexpected behavior due to aliasing.

Because tuples are immutable, they don't provide methods like `sort` and `reverse`, which modify existing lists. However Python provides the built-in functions `sorted` and `reversed`, which take any sequence as a parameter and return a new sequence with the same elements in a different order.

10.9: Debugging

Lists, dictionaries and tuples are known generically as *data structures*; in this chapter we are starting to see compound data structures, like lists of tuples, and dictionaries that contain tuples as keys and lists as values. Compound data structures are useful, but they are prone to what I call *shape errors*; that is, errors caused when a data structure has the wrong type, size, or composition, or perhaps you write some code and forget the shape of your data and introduce an error.

For example, if you are expecting a list with one integer and I give you a plain old integer (not in a list), it won't work.

When you are debugging a program, and especially if you are working on a hard bug, there are four things to try:

reading

Examine your code, read it back to yourself, and check that it says what you meant to say.

running

Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to spend some time to build scaffolding.

ruminating

Take some time to think! What kind of error is it: syntax, runtime, semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared?

retreating

At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works and that you understand. Then you can start rebuilding.

Beginning programmers sometimes get stuck on one of these activities and forget the others. Each activity comes with its own failure mode.

For example, reading your code might help if the problem is a typographical error, but not if the problem is a conceptual misunderstanding. If you don't understand what your program does, you can read it 100 times and never see the error, because the error is in your head.

Running experiments can help, especially if you run small, simple tests. But if you run experiments without thinking or reading your code, you might fall into a pattern I call "random walk programming", which is the process of making random changes until the program does the right thing. Needless to say, random walk programming can take a long time.

You have to take time to think. Debugging is like an experimental science. You should have at least one hypothesis about what the problem is. If there are two or more possibilities, try to think of a test that would eliminate one of them.

Taking a break helps with the thinking. So does talking. If you explain the problem to someone else (or even to yourself), you will sometimes find the answer before you finish asking the question.

But even the best debugging techniques will fail if there are too many errors, or if the code you are trying to fix is too big and complicated. Sometimes the best option is to retreat, simplifying the program until you get to something that works and that you understand.

Beginning programmers are often reluctant to retreat because they can't stand to delete a line of code (even if it's wrong). If it makes you feel better, copy your program into another file before you start stripping it down. Then you can paste the pieces back in a little bit at a time.

Finding a hard bug requires reading, running, ruminating, and sometimes retreating. If you get stuck on one of these activities, try the others.

10.E: Tuples (Exercises)

Exercise 1: Revise a previous program as follows: Read and parse the "From" lines and pull out the addresses from the line. Count the number of messages from each person using a dictionary.

After all the data has been read, print the person with the most commits by creating a list of (count, email) tuples from the dictionary. Then sort the list in reverse order and print out the person who has the most commits.

```
Sample Line:
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008

Enter a file name: mbox-short.txt
cwen@iupui.edu 5

Enter a file name: mbox.txt
zqian@umich.edu 195
```

Exercise 2: This program counts the distribution of the hour of the day for each of the messages. You can pull the hour from the "From" line by finding the time string and then splitting that string into parts using the colon character. Once you have accumulated the counts for each hour, print out the counts, one per line, sorted by hour as shown below.

Sample Execution:

```
python timeofday.py
Enter a file name: mbox-short.txt
04 3
06 1
07 1
09 2
10 3
11 6
14 1
15 2
16 4
17 2
18 1
19 1
```

Exercise 3: Write a program that reads a file and prints the *letters* in decreasing order of frequency. Your program should convert all the input to lower case and only count the letters a-z. Your program should not count spaces, digits, punctuation, or anything other than the letters a-z. Find text samples from several different languages and see how letter frequency varies between languages. Compare your results with the tables at Wikipedia.org/wiki/Letter_frequencies.

---

1. Fun fact: The word "tuple" comes from the names given to sequences of numbers of varying lengths: single, double, triple, quadruple, quituple, sextuple, septuple, etc.↩
2. Python does not translate the syntax literally. For example, if you try this with a dictionary, it will not work as might expect.↩

10.G: Tuples (Glossary)

comparable

A type where one value can be checked to see if it is greater than, less than, or equal to another value of the same type. Types which are comparable can be put in a list and sorted.

data structure

A collection of related values, often organized in lists, dictionaries, tuples, etc.

DSU

Abbreviation of "decorate-sort-undecorate", a pattern that involves building a list of tuples, sorting, and extracting part of the result.

gather

The operation of assembling a variable-length argument tuple.

hashable

A type that has a hash function. Immutable types like integers, floats, and strings are hashable; mutable types like lists and dictionaries are not.

scatter

The operation of treating a sequence as a list of arguments.

shape (of a data structure)

A summary of the type, size, and composition of a data structure.

singleton

A list (or other sequence) with a single element.

tuple

An immutable sequence of elements.

tuple assignment

An assignment with a sequence on the right side and a tuple of variables on the left. The right side is evaluated and then its elements are assigned to the variables on the left.