

CS303E: Elements of Computers and Programming

Tuples, Sets, Dictionaries

Dr. Bill Young
Department of Computer Science
University of Texas at Austin
© William D. Young, All rights reserved.

Last updated: August 29, 2024 at 12:56

Tuples

Another useful data type in Python is **tuples**. Tuples are like immutable lists of fixed size, but allow faster access than lists.

```
>>> tuple() # create an empty tuple
()
>>> t1 = () # special syntax
>>> t1
()
>>> t2 = tuple( [1, 2, 3] ) # 3-tuple from list
>>> t2
(1, 2, 3)
>>> (1) # not considered a tuple
1
>>> t3 = tuple([1]) # force 1-tuple from list
>>> t3
(1,)
>>> t4 = (2,) # note odd syntax
>>> t4
(2,)
```

Sequence Operations for Tuples

Tuples, like strings and list, are sequences and inherit various functions from sequences. Like strings, but unlike lists, they are immutable.

Function	Description
<code>x in t</code>	<code>x</code> is in tuple <code>t</code>
<code>x not in t</code>	<code>x</code> is not in tuple <code>t</code>
<code>t1 + t2</code>	concatenates two tuples
<code>t * n</code>	repeat tuple <code>t</code> <code>n</code> times
<code>t[i]</code>	<code>i</code> th element of tuple (0-based)
<code>t[i:j]</code>	slice of tuple <code>t</code> from <code>i</code> to <code>j-1</code>
<code>len(t)</code>	number of elements in <code>t</code>
<code>min(t)</code>	minimum element of <code>t</code>
<code>max(t)</code>	maximum element of <code>t</code>
<code>sum(t)</code>	sum of elements in <code>t</code>
for loop	traverse elements of tuple
<code><</code> , <code><=</code> , <code>></code> , <code>>=</code>	compares two tuples
<code>==</code> , <code>!=</code>	compares two tuples

Some Tuple Examples

```
>>> t1 = tuple([ 1, "red", 2.3 ]) # tuple from list
>>> 'red' in t1
True
>>> 'green' in t1
False
>>> t1 + ("green", 4.5 ) # tuple concatenation
(1, 'red', 2.3, 'green', 4.5)
>>> t2 = t1 * 3 # repeat tuple
>>> t2
(1, 'red', 2.3, 1, 'red', 2.3, 1, 'red', 2.3)
>>> t2[3] # indexing
1
>>> len(t2) # using len
9
>>> min(t2) # using min
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between 'str' and 'int'
>>> t3 = tuple( [ x for x in range(11) ] )
>>> t3
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

If you want to manipulate (e.g., shuffle) a tuple, you can convert to a list first, and then back to a tuple.

```
>>> t3
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> lst = list( t3 )
>>> lst
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> import random
>>> lst2 = random.shuffle( lst ) # a common error!
>>> print(lst2)                 # what happened?
None
>>> random.shuffle( lst )      # shuffles in place
>>> lst
[1, 4, 7, 3, 5, 0, 6, 9, 8, 2, 10]
>>> tuple(lst)
(1, 4, 7, 3, 5, 0, 6, 9, 8, 2, 10)
```

Functions can return tuples just as they can return other values. Specifically, if they return multiple values, they are really returning a tuple.

In file Tuple.py:

```
def MultiValues (x):
    return x + 4, x - 4, x ** 2 # 3-tuple
```

```
>>> from Tuple import *
>>> MultiValues( 9 ) # returns 3-tuple
(13, 5, 81)
>>> t1 = MultiValues( 9 ) # save as 3-tuple
>>> t1[0]
13
>>> x, y, z = MultiValues( 9 ) # save separately
>>> print( "x:", x, "y:", y, "z:", z )
x: 13 y: 5 z: 81
```

Sets are similar to lists except:

- sets don't store duplicate elements;
- sets are not ordered.

```
>>> s1 = set() # empty set
>>> s1
set() # notice odd syntax
>>> s1 is {} # {} is a dictionary,
False # not a set
>>> type({})
<class 'dict'>
>>> type(set())
<class 'set'>
>>> s2 = set([1, 2, 2, 4, 3]) # set from list
>>> s2
{1, 2, 3, 4} # no duplicates
>>> set("abcda") # set from string
{'d', 'a', 'c', 'b'}
>>> {'d', 'a', 'c', 'b'} == {'a', 'c', 'b', 'd'}
True # order doesn't matter
>>> t = ("abc", 4, 2.3)
>>> set(t) # set from tuple
{2.3, 'abc', 4}
```

The following sequence functions are available on sets.

Function	Description
<code>x in s</code>	<code>x</code> is in set <code>s</code>
<code>x not in s</code>	<code>x</code> is not in set <code>s</code>
<code>len(s)</code>	number of elements in <code>s</code>
<code>min(s)</code>	minimum element of <code>s</code>
<code>max(s)</code>	maximum element of <code>s</code>
<code>sum(s)</code>	sum of elements in <code>s</code>
<code>for loop</code>	traverse elements of set

```
>>> s = {1, 2, "red", "green", 3.5 }
>>> s
{1, 2, 3.5, 'green', 'red'} # order doesn't matter
>>> 2 in s
True
>>> 3 in s
False
>>> len( s )
5
>>> min( s ) # items must be comparable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between 'str' and 'int'
>>> min( { -2, 17, 9, 4 } )
-2
>>> max( { -2, 17, 9, 4 } )
17
>>> sum( { -2, 17, 9, 4 } )
28
>>> for i in s: print( i, end = " " )
...
1 2 3.5 green red >>>
```

Like lists, sets are mutable. These two methods alter the set.

Function	Description
<code>s.add(e)</code>	add e to set s
<code>s.remove(e)</code>	remove e from set s

```
>>> s = set() # create empty set
>>> s
set()
>>> s.add(2.5) # changes s
>>> s.add("red") # changes s
>>> s.add(1) # changes s
>>> s.add("red") # change?
>>> s
{1, 2.5, 'red'}
>>> s.remove("green") # item must appear
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'green'
>>> s.remove("red") # changes s
>>> s
{1, 2.5}
```

Subset and Superset

s_1 is a *subset* of s_2 if every element of s_1 is also an element of s_2 .
If s_1 is a subset of s_2 , then s_2 is a *superset* of s_1 .

Function	Description
<code>s1.issubset(s2)</code>	s_1 is a subset of s_2
<code>s2.issuperset(s1)</code>	s_1 is a subset of s_2

Notice that s is always a subset and superset of itself.

```
>>> s1 = { 2, 3, 5, 7 }
>>> s2 = { 2, 5, 7 }
>>> s2.issubset(s1)
True
>>> s1.issuperset(s2)
True
>>> s1.issubset(s1)
True
>>> s2.add(8)
>>> s2
{8, 2, 5, 7}
>>> s2.issubset(s1)
False
```

Subset: Alternate Syntax

Function	Description
<code>s1 <= s2</code>	s_1 is a subset of s_2
<code>s1 < s2</code>	s_1 is a proper subset of s_2
<code>s2 >= s1</code>	s_2 is a superset of s_1
<code>s2 > s1</code>	s_2 is a proper superset of s_1

s_1 is a *proper* subset of s_2 if s_1 is a subset of s_2 , but not equal to s_2 .

```
>>> s1 = { 1, 2, 3 }
>>> s2 = { 0, 1, 2, 3, 4 }
>>> s1 < s2 # is s1 a proper subset of s2
True
>>> s1 <= s2 # is s1 a subset of s2
True
>>> s1 < s1 # is s1 a proper subset of itself
False
>>> s1 <= s1 # is s1 a subset of itself
True
>>> s2 > s1 # is s2 a proper superset of s1
True
```

The following operations take two sets and return a new set.

Function	Alternate Syntax	Description
<code>s1.union(s2)</code>	<code>s1 s2</code>	elements in s1 or s2
<code>s1.intersection(s2)</code>	<code>s1 & s2</code>	elements in both s1 and s2
<code>s1.difference(s2)</code>	<code>s1 - s2</code>	elements in s1 but not in s2
<code>s1.symmetric_difference(s2)</code>	<code>s1 ^ s2</code>	elements in s1 or s2, but not both

```
>>> s1 = { 1, 2, 3 }
>>> s2 = { 1, 3, 5, 7 }
>>> s1.union(s2)           # new set
{1, 2, 3, 5, 7}
>>> s2.union(s1)           # new set, commutes
{1, 2, 3, 5, 7}
>>> s1 | s2                # alternate syntax
{1, 2, 3, 5, 7}
```

```
>>> s1 = { 1, 2, 3 }
>>> s2 = { 1, 3, 5, 7 }
>>> s1.intersection(s2)   # new set
{1, 3}
>>> s1 & s2                # alternate syntax
{1, 3}
>>> s1.difference(s2)     # new set
{2}
>>> s2.difference(s1)     # not commutative
{5, 7}
>>> s1 - s2 == s2 - s1
False
>>> s1.symmetric_difference(s2) # new set
{2, 5, 7}
>>> s1 ^ s2                # alternate syntax
{2, 5, 7}
>>> s2 ^ s1                # commutes
{2, 5, 7}
```

Set Example: Count Keywords

In file `CountKeywords.py`:

```
import os.path

def CountKeywordsWithSet():
    """ Count the number of occurrence of keywords in a
        Python source code file specified by the user. """
    keywords = \
        { "and", "as", "assert", "break", "class",
          "continue", "def", "del", "elif", "else",
          "except", "False", "finally", "for", "from",
          "global", "if", "import", "in", "is", "lambda",
          "nonlocal", "None", "not", "or", "pass", "raise",
          "return", "True", "try", "while", "with", "yield" }

    # Accept a filename from the user.
    filename = input("Enter a filename: ").strip()
    # Check that the file exists.
    if not os.path.isfile( filename ):
        print( "File", filename, "does not exist." )
        return
    infile = open(filename, "r")
```

Code continues on next slide.

Set Example: Count Keywords

```
    # Read the file line by line, counting keywords.
    count = 0
    keywordsFound = set()
    line = infile.readline()
    while line:
        words = line.split()
        # Record keywords found in set keywordsFound.
        for word in words:
            if word in keywords:
                count += 1
                keywordsFound.add( word )
        line = infile.readline()
    # Print the results.
    print("Found", count, "keyword occurrences in file",
          filename)
    print("Keywords found:", keywordsFound )

CountKeywordsWithSet()
```

```
> python CountKeywords.py
Enter a filename: CountKeywords.py
Found 13 keyword occurrences in file CountKeywords.py
Keywords found: {'def', 'import', 'not', 'from', 'in', 'for',
, 'if', 'return'}
```

This program could be improved. [Can you see how?](#)

```
> python CountKeywords.py
Enter a filename: CountKeywords.py
Found 13 keyword occurrences in file CountKeywords.py
Keywords found: {'def', 'import', 'not', 'from', 'in', 'for',
, 'if', 'return'}
```

This program could be improved. [Can you see how?](#)

Since we split on whitespace, this will miss keywords that have adjacent punctuation like "True:"

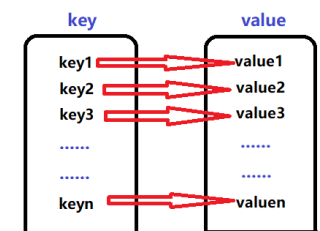


A Python **dictionary** stores a set of key/value pairs. It enables very fast retrieval, deletion and updating of values using the keys.

```
squares = { 2 : 4, 3 : 9, 4 : 16, 5 : 25 }
```

Imagine a regular dictionary; associated with each word is a definition.

The word is the **key**, and the definition is the **value**.



The most fundamental operation is being able (quickly) to look up the value associated with the key.

Use curly braces ({}) to denote a dictionary (and a set).

To add (or change) an item in a dictionary, use the syntax:

```
dictionaryName[key] = value
```

To retrieve the value associated with key, use:

```
dictionaryName[key]
```

To delete a key/value from the dictionary:

```
del dictionaryName[key]
```

```
>>> midterms = {} # empty dictionary
>>> midterms['Susie'] = 80 # add 'Susie' : 80
>>> midterms['Frank'] = 87 # add 'Frank' : 87
>>> midterms['Albert'] = 56 # add 'Albert': 56
>>> midterms
{'Susie': 80, 'Frank': 87, 'Albert': 56}
>>> midterms['Susie'] = 82 # change Susie's grade
>>> midterms['Charles'] = 79 # add 'Charles': 79
```

```
>>> midterms # show midterms
{'Susie': 82, 'Frank': 87, 'Albert': 56, 'Charles': 79}
>>> midterms['Frank'] # what's Frank's grade
87
>>> midterms['Susie'] = 'dropped' # record Susie dropped
>>> midterms
{'Susie': 'dropped', 'Frank': 87, 'Albert': 56, 'Charles': 79}
>>> midterms['Susie'] # what's Susie's grade
'dropped'
>>> del midterms['Albert'] # delete Albert's record
>>> midterms
{'Susie': 'dropped', 'Frank': 87, 'Charles': 79}
>>> del midterms['Tony'] # delete Tony's record
Traceback (most recent call last): # Tony's not in the
  File "<stdin>", line 1, in <module> # class
KeyError: 'Tony'
```

As of Python 3.7, dictionaries are kept in the order items were entered. However, it's a bad idea to rely on the order.

Looping Over a Dictionary

The most common way to iterate over a dictionary is to loop over the keys.

```
for key in dictionaryName:
    < body >
```

```
>>> midterms = {'Susie': 'dropped', 'Frank': 87, 'Charles': 79}
>>> for key in midterms:
...     print( key, ":", midterms[key] )
...
Susie : dropped
Frank : 87
Charles : 79
```

Notice that dictionary keys (like sets) are not ordered. Two dictionaries are equal if they contain the same pairs:

```
>>> {'Susie':14, 'Frank':87} == {'Frank':87, 'Susie':14}
True
```

Dictionary Functions

The following sequence functions work for dictionaries:

Function	Description
key in dict	key is in the dict
key not in dict	key is not in dict
len(dict)	number of key/value pairs in dict
min(dict)	minimum key in dict, if comparable
max(dict)	maximum key in dict, if comparable
sum(dict)	sum of keys in dict, if summable
for key in dict	traverse dictionary
==, !=	compares two dictionaries

```

>>> dict1 = {'Susie':87, 'Frank':78, 'Charles':90}
>>> 'Susie' in dict1
True
>>> 'susie' in dict1      # case matters
False
>>> 'frank' not in dict1
True
>>> len( dict1 )      # number of key/value pairs
3
>>> min( dict1 )      # minimum key
'Charles'
>>> max( dict1 )      # maximum key
'Susie'
>>> sum( dict1 )      # only if keys are summable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported type(s) for +: 'int' and 'str'
>>> squares = {2:4, 3:9, 4:16, 5:25, 6:36}
>>> sum(squares)      # sums keys, not values
20

```

These are methods from class dict. Dictionaries are mutable; the final three change d.

Function	Description
d.keys()	return the keys of d as a tuple*
d.values()	return the values of d as a tuple*
d.items()	return the key/value pairs from d as a tuple*
d.get(key)	return the value for the key, same as d[key]
d.clear()	delete all items in d
d.pop(key)	remove item with key and return the value
d.popitem()	remove a randomly selected item and return that key/value pair

Why wouldn't it make sense for d.popitem() to return the last item of d as in a list?

*It's not exactly a tuple. See the following slide on "Dictionary Views."

```

>>> dict1 = {'Susie':87, 'Frank':78, 'Charles':90}
>>> dict1.keys()
dict_keys(['Susie', 'Frank', 'Charles'])
>>> dict1.values()
dict_values([87, 78, 90])
>>> dict1.items()
dict_items([('Susie', 87), ('Frank', 78), ('Charles', 90)])
>>> dict1.get('Frank')
78
>>> dict1.pop('Charles')
90
>>> dict1
{'Susie': 87, 'Frank': 78}
>>> dict1['Bernard'] = 92
>>> dict1
{'Susie': 87, 'Frank': 78, 'Bernard': 92}
>>> dict1.popitem()
('Bernard', 92)
>>> dict1.popitem()
('Frank', 78)
>>> dict1.clear()
>>> dict1
{}

```

```

>>> dict1.keys()
dict_keys(['Susie', 'Frank', 'Charles'])
>>> dict1.values()
dict_values([87, 78, 90])
>>> dict1.items()
dict_items([('Susie', 87), ('Frank', 78), ('Charles', 90)])
>>>

```

Those odd types dict_keys, dict_values, dict_items are *dictionary views*. They're tied to the dictionary; if you change the dictionary, the views change even if you try to save them into a variable.

```

>>> dict.items()
dict_items([('Susie', 50), ('Frank', 88)])
>>> x = dict.items()
>>> x
dict_items([('Susie', 50), ('Frank', 88)])
>>> dict['Susie'] = 25
>>> x
dict_items([('Susie', 25), ('Frank', 88)])
>>>

```

In file CountKeywords.py:

```
def CountKeywordsWithDictionary():
    """ Count the number of occurrence of keywords in a
        Python source code file specified by the user,
        using a dictionary to record the counts."""
    keywords = \
        { "and", "as", "assert", "break", "class",
          "continue", "def", "del", "elif", "else",
          "except", "False", "finally", "for", "from",
          "global", "if", "import", "in", "is", "lambda",
          "nonlocal", "None", "not", "or", "pass", "raise",
          "return", "True", "try", "while", "with", "yield" }

    # Accept a filename from the user.
    filename = input("Enter a filename: ").strip()
    # Check that the file exists.
    if not os.path.isfile( filename ):
        print( "File", filename, "does not exist.")
        return
    infile = open(filename, "r")
```

Code continues on next slide:

```
keywordsFound = {}
line = infile.readline()
while line:
    words = line.split()
    for word in words:
        # Is word is a keyword?
        if word in keywords:
            # Is it already in the dictionary?
            if word in keywordsFound:
                # If so, increment the counter
                keywordsFound[word] += 1
            else:
                # Otherwise, start counter at 1.
                keywordsFound[word] = 1
    line = infile.readline()
# How many total keywords were found?
totalCount = sum( keywordsFound.values() )
# Print the results.
print("Found", totalCount, "keyword occurrences in", \
      filename)
print("Keywords found:")
for key in keywordsFound:
    print(" ", key + ":", keywordsFound[key] )
```

Running the Code

```
>>> CountKeywordsWithDictionary()
Enter a filename: CountKeywords.py
Found 33 keyword occurrences in CountKeywords.py
Keywords found:
import: 1
def: 2
in: 10
from: 2
if: 5
not: 4
return: 2
and: 2
as: 2
for: 3
```

By the way, the reason the counts don't match what we got with CountKeywordsWithSet is because I added the code for CountKeywordsWithDictionary to the file.



Next step: Recursion.