STATS 701
Data Analysis using Python

Lecture 5: Tuples

Tuples

Similar to a list, in that it is a sequence of values

But unlike lists, tuples are immutable

Because they are immutable, they are hashable

So we can use tuples where we wanted to key on a list

Documentation:

https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences

https://docs.python.org/3/library/stdtypes.html#tuples

Creating Tuples

```
1 t = 1,2,3,4,5
2 t
```

```
(1, 2, 3, 4, 5)
```

```
1 t = (1,2,3,4,5)
2 t
```

```
(1, 2, 3, 4, 5)
```

```
1 t = 'cat',
2 t
```

```
('cat',)
```

```
1 t = ('cat')
2 t
```

```
'cat'
```

Tuples created either with “comma notation”, optional parentheses.

Python always displays tuples with parentheses.

Creating a tuple of one element requires a trailing comma. Failure to include this comma, even with parentheses, yields… not a tuple.

Creating Tuples

```
1 t1 = tuple()
2 t1
```

```
()
```

```
1 t2 = tuple(range(5))
2 t2
```

```
(0, 1, 2, 3, 4)
```

```
1 t3 = tuple('goat')
2 t3
```

```
('g', 'o', 'a', 't')
```

```
1 tuple([[1,2,3],[4,5,6]])
```

```
([1, 2, 3], [4, 5, 6])
```

```
1 print(type(t2))
```

```
<class 'tuple'>
```

Can also create a tuple using the `tuple()` function, which will cast any sequence to a tuple whose elements are those of of the sequence.

Tuples are Sequences

```
1 t = ('a', 'b', 'c', 'd', 'e')
2 t[0]
```

```
'a'
```

As sequences, tuples support indexing, slices, etc.

```
1 t[1:4]
```

```
('b', 'c', 'd')
```

```
1 t[-1]
```

```
'e'
```

And of course, sequences have a length.

```
1 len(t)
```

```
5
```

Reminder: sequences support all the operations listed here: https://docs.python.org/3.3/library/stdtypes.html#typesseq

Tuple Comparison

Tuples support comparison, which works analogously to string ordering.

```
1 (1,2,3) < (2,2,3)
```

```
True
```

```
1 (2,2,20) <= (2,2,2)
```

```
False
```

0-th elements are compared. If they are equal, go to the 1-th element, etc.

```
1 (1,2,3) < (1,2,3,4)
```

```
True
```

Just like strings, the “prefix” tuple is ordered first.

```
1 ('cat','dog','goat') > ('dog', 'cat', 'goat')
```

```
False
```

```
1 (1,'cat',(1,2,3)) > (0,'bird',(1,2,0))
```

```
True
```

Tuple comparison is element-wise, so we only need that each element-wise comparison is allowed by Python.

Tuples are Immutable

```
1 fruits = ('apple', 'banana', 'orange', 'kiwi')
2 fruits[2] = 'grapefruit'
```

Tuples are immutable, so changing an entry is not permitted.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-48-c40a1905a6e9> in <module>()
      1 fruits = ('apple', 'banana', 'orange', 'kiwi')
----> 2 fruits[2] = 'grapefruit'

TypeError: 'tuple' object does not support item assignment
```

As with strings, have to make a new assignment to the variable.

```
1 fruits = fruits[0:2] + ('grapefruit',) + fruits[3:]
2 fruits
```

```
('apple', 'banana', 'grapefruit', 'kiwi')
```

```
1 fruits = fruits[0:2] + 'grapefruit', + fruits[3:]
```

Note: even thought `'grapefruit',` is a tuple, Python doesn't know how to parse this line. Use parentheses!

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-50-f62749483e65> in <module>()
----> 1 fruits = fruits[0:2] + 'grapefruit', + fruits[3:]

TypeError: can only concatenate tuple (not "str") to tuple
```

Useful trick: tuple assignment

```
1 a = 10
2 b = 5
3 print(a, b)
```

```
10 5
```

Tuples in Python allow us to make many variable assignments at once. Useful tricks like this are sometimes called **syntactic sugar** (though some might argue that tuple assignment isn't *technically* an example of such). https://en.wikipedia.org/wiki/Syntactic_sugar

```
tmp = a
a = b
b = tmp
print(a, b)
```

```
10 5
```

Common pattern: swap the values of two variables.

```
1 a = 10
2 b = 5
3 (a,b) = (b,a)
4 print(a, b)
```

```
5 10
```

This line achieves the same end, but in a single assignment statement instead of three, and without the extra variable `tmp`.

Useful trick: tuple assignment

```
1 (x,y,z) = (2*'cat', 0.57721, [1,2,3])
2 (x,y,z)
```

```
('catcat', 0.57721, [1, 2, 3])
```

Tuple assignment requires one variable on the left for each expression on the right.

```
1 (x,y,z) = ('a','b','c','d')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-68-e118c50f83dd> in <module>()
----> 1 (x,y,z) = ('a','b','c','d')

ValueError: too many values to unpack (expected 3)
```

If the number of variables doesn't match the number of expressions, that's an error.

```
1 (x,y,z) = ('a','b')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-69-875f95cea434> in <module>()
----> 1 (x,y,z) = ('a','b')

ValueError: not enough values to unpack (expected 3, got 2)
```

Useful trick: tuple assignment

```
1 email = 'klevin@umich.edu'
2 email.split('@')
```

```
['klevin', 'umich.edu']
```

The `string.split()` method returns a list of strings, obtained by splitting the calling string on the characters in its argument.

```
1 (user,domain) = email.split('@')
2 user
```

```
'klevin'
```

Tuple assignment works so long as the right-hand side is **any** sequence, provided the number of variables matches the number of elements on the right. Here, the right-hand side is a list, `['klevin', 'umich.edu']`.

```
1 domain
```

```
'umich.edu'
```

```
1 (x,y,z) = 'cat'
2 print(x, y, z)
```

```
c a t
```

A string is a sequence, so tuple assignment is allowed. Sequence elements are characters, and indeed, `x`, `y` and `z` are assigned to the three characters in the string.

Tuples as Return Values

```
1 import random
2 def five_numbers(t):
3     t.sort()
4     n = len(t)
5     return (t[0],  t[n//4],  t[n//2], t[(3*n)//4], t[-1])
6 five_numbers([1,2,3,4,5,6,7])
```

```
(1, 2, 4, 6, 7)
```

This function takes a list of numbers and returns a tuple summarizing the list.
https://en.wikipedia.org/wiki/Five-number_summary

```
1 randnumlist = [random.randint(1,100) for x in range(60)]
2 (mini,lowq,med,upq,maxi) = five_numbers(randnumlist)
3 (mini,lowq,med,upq,maxi)
```

```
(3, 27, 54, 73, 98)
```

Test your understanding: what does this list comprehension do?

Tuples as Return Values

More generally, sometimes you want more than one return value

```
1 t = divmod(13,4)
2 t
```

```
(3, 1)
```

```
1 (quotient,remainder) = divmod(13,4)
2 quotient
```

```
3
```

```
1 remainder
```

```
1
```

`divmod` is a Python built-in function that takes a pair of numbers and outputs the quotient and remainder, as a tuple. Additional examples can be found here: https://docs.python.org/3/library/functions.html

Useful trick: variable-length arguments

```
1 def my_min( *args ):
2     return min(args)
3 my_min(1,2,3)
```

```
1
```

```
1 my_min(4,5,6,10)
```

```
4
```

```
1 def print_all( *args ):
2     print(args)
3 print_all('cat', 'dog', 'bird')
```

```
('cat', 'dog', 'bird')
```

```
1 print_all()
```

```
()
```

A parameter name prefaced with * **gathers** all arguments supplied to the function into a tuple.

Note: this is also one of several ways that one can implement optional arguments, though we'll see better ways later in the course.

Gather and Scatter

The opposite of the gather operation is **scatter**

```
1 t = (13, 4)
2 divmod(t)
```

`divmod` takes two arguments, so this is an error.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-106-c7c0a10eef7e> in <module>()
      1 t = (13, 4)
----> 2 divmod(t)

TypeError: divmod expected 2 arguments, got 1
```

```
1 divmod(*t)
```

```
(3, 1)
```

Instead, we have to “untuple” the tuple, using the **scatter** operation. This makes the elements of the tuple into the arguments of the function.

```
1 *t
```

```
  File "<ipython-input-109-f9912a2ca07d>", line 1
    *t
     ^
SyntaxError: can't use starred expression here
```

Note: gather/scatter only works in certain contexts (e.g., for function arguments).

Combining lists: `zip`

Python includes a number of useful functions for combining lists and tuples

```
1 t1 = ['apple', 'orange', 'banana', 'kiwi']
2 t2 = [1, 2, 3, 4]
3 zip(t1,t2)
```

```
<zip at 0x10c95d5c8>
```

`zip()` returns a zip object, which is an **iterator** containing as its elements tuples formed from its arguments.
https://docs.python.org/3/library/functions.html#zip

```
1 for tup in zip(t1,t2):
2     print(tup)
```

```
('apple', 1)
('orange', 2)
('banana', 3)
('kiwi', 4)
```

Iterators are, in essence, objects that support for-loops. All sequences are iterators. Iterators support, crucially, a method `__next__()`, which returns the “next element”. We’ll see this in more detail later in the course.
https://docs.python.org/3/library/stdtypes.html#iterator-types

Combining lists: `zip`

`zip()` returns a zip object, which is an **iterator** containing as its elements tuples formed from its arguments.
https://docs.python.org/3/library/functions.html#zip

```
1 for tup in zip(['a','b','c'],[1,2,3,4]):
2     print(tup)
```

```
('a', 1)
('b', 2)
('c', 3)
```

Given arguments of different lengths, `zip` defaults to the shortest one.

```
1 for tup in zip(['a','b','c','d'],[1,2,3]):
2     print(tup)
```

```
('a', 1)
('b', 2)
('c', 3)
```

`zip` takes any number of arguments, so long as they are all **iterable**. Sequences are iterable.

```
1 for tup in zip([1,2,3],['a','b','c'],'xyz'):
2     print(tup)
```

```
(1, 'a', 'x')
(2, 'b', 'y')
(3, 'c', 'z')
```

Iterables are, essentially, objects that can *become* iterators. We'll see the distinction later in the course.
https://docs.python.org/3/library/stdtypes.html#typeiter

Combining lists: `zip`

```
1 def count_matches(s, t):
2     cnt = 0
3     for (a,b) in zip(s,t):
4         if a==b:
5             cnt += 1
6     return( cnt )
7 count_matches([1,1,2,3,5],[1,2,3,4,5])
```

```
2
```

`zip` is especially useful for iterating over several lists in lockstep.

```
1 count_matches([1,2,3,4,5],[1,2,3])
```

Test your understanding: what should this return?

Combining lists: `zip`

```
1 def count_matches(s, t):
2     cnt = 0
3     for (a,b) in zip(s,t):
4         if a==b:
5             cnt += 1
6     return( cnt )
7 count_matches([1,1,2,3,5],[1,2,3,4,5])
```

```
2
```

`zip` is especially useful for iterating over several lists in lockstep.

```
1 count_matches([1,2,3,4,5],[1,2,3])
```

```
3
```

Test your understanding: what should this return?

Related function: `enumerate()`

```
1 for t in enumerate('goat'):
2     print(t)
```

```
(0, 'g')
(1, 'o')
(2, 'a')
(3, 't')
```

```
1 s = 'goat'
2 for i in range(len(s)):
3     print((i,s[i]))
```

```
(0, 'g')
(1, 'o')
(2, 'a')
(3, 't')
```

`enumerate` returns an **enumerate object**, which is an iterator of (index,element) pairs. It is a more graceful way of performing the pattern below, which we've seen before.
https://docs.python.org/3/library/functions.html#enumerate

Dictionaries revisited

```
1 hist = {'cat':3,'dog':12,'goat':18}
2 hist.items()
```

```
dict_items([('cat', 3), ('dog', 12), ('goat', 18)])
```

```
1 for (k,v) in hist.items():
2     print(k, ':', v)
```

```
cat : 3
dog : 12
goat : 18
```

`dict.items()` returns a dict_items object, an iterator whose elements are (key,value) tuples.

```
1 d = dict([(0,'zero'),(1,'one'),(2,'two')])
2 d
```

```
{0: 'zero', 1: 'one', 2: 'two'}
```

Conversely, we can create a dictionary by supplying a list of (key,value) tuples.

```
1 dict( zip('cat','dog'))
```

```
{'a': 'o', 'c': 'd', 't': 'g'}
```

Tuples as Keys

```
1 name2umid = {('Einstein','Albert'): 'aeinstein',
2  ('Noether','Emmy'): 'enoether',
3  ('Shannon','Claude'): 'cshannon',
4  ('Fan','Ky'): 'kyfan'}
5 name2umid
```

```
{('Einstein', 'Albert'): 'aeinstein',
 ('Fan', 'Ky'): 'kyfan',
 ('Noether', 'Emmy'): 'enoether',
 ('Shannon', 'Claude'): 'cshannon'}
```

In (most) Western countries, the family name is said last (hence “last name”), but it is frequently useful to key on this name before keying on a given name.

```
1 name2umid[('Einstein','Albert')]
```

```
'aeinstein'
```

```
1 sparsemx = dict()
2 sparsemx[(1,4)] = 1
3 sparsemx[(3,5)] = 1
4 sparsemx[(12,13)] = 2
5 sparsemx[(11,13)] = 3
6 sparsemx[(19,13)] = 5
7 sparsemx
```

```
{(1, 4): 1, (3, 5): 1, (11, 13): 3, (12, 13): 2, (19, 13): 5}
```

Keying on tuples is especially useful for representing sparse structures. Consider a 20-by-20 matrix in which most entries are zeros. Storing all the entries requires 400 numbers, but if we only record the entries that are nonzero...

Data Structures: Lists vs Tuples

Use a **list** when:

- Length is not known ahead of time and/or may change during execution
- Frequent updates are likely

Use a **tuple** when:

- The set is unlikely to change during execution
- Need to key on the set (i.e., require immutability)
- Want to perform multiple assignment or for use in variable-length arg list

Most code you see will use lists, because mutability is very useful!

Readings (this lecture)

Required:

Downey Chapter 12 or Severance Chapter 10

Recommended:

Downey Chapter 13

Python documentation on tuples

https://docs.python.org/3/library/stdtypes.html#tuples

https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences

Readings (next lecture)

Required:

Downey Chapter 14 or Severance Chapter 7

Python File I/O Documentation:

https://docs.python.org/3/tutorial/inputoutput.html

Handling Errors and Exceptions:

https://docs.python.org/3/tutorial/errors.html

Recommended:

Python pickle module:

https://docs.python.org/3/library/pickle.html#module-pickle