**Tuples**

In Python tuples are very similar to lists, however, unlike lists they are *immutable* meaning they can not be changed. You would use tuples to present things that shouldn't be changed, such as days of the week, or dates on a calendar.

In this section, we will get a brief overview of the following:

1.) Constructing Tuples

2.) Basic Tuple Methods

3.) Immutability

4.) When to Use Tuples

**Constructing Tuples**

The construction of a tuples use () with elements separated by commas. once we create tuple we cannot modify its element hence can not perform operations like append(), extend(), insert(),remove(), pop(), clear()

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. 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 **=** ()

t **=** (1,2,3,4,"Mary",23.4)

print(t[0])

t **=** ("Sunday","Monday")

​

​

​

1

lst **=** [1,2,3,4]

t**=**tuple(lst)

print(t)

​

print(tuple(range(10)))

st **=** "Asha"

t **=** tuple(st)

print(t)

​

t **=** (1,2,3,4)

print(type(t))

t **=** 1,2,3,4

print(type(t))

t **=** 1

t **=** 1,2,3

​

​

t1,t2,t3 **=** t

print(t1,t2,t3)

t **=** 1,

print(type(t))

(1, 2, 3, 4)

(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

('A', 's', 'h', 'a')

<class 'tuple'>

<class 'tuple'>

1 2 3

<class 'tuple'>

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

t1 **=** ('a',)

print(type(t1))

<class 'tuple'>

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

​

t2 **=** ('a')

print(type(t2))

​

<class '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:*

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

t **=** tuple('lupins')

print(t)

​

t **=** list('lupins')

print(t)

​

('l', 'u', 'p', 'i', 'n', 's')

['l', 'u', 'p', 'i', 'n', 's']

*# 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)

​

​

​

Out[9]:

True

(0, 1, 2000000) **<** (0, 3, 4) *#1 is less than 3 it will not consider subswquent elements 200000 even though it is big*

Out[10]:

True

*# Create a tuple*

t **=** (1,2,3)

t

Out[12]:

(1, 2, 3)

*# Check len just like a list*

len(t)

Out[13]:

3

*# Can also mix object types*

t **=** ('one','one',2)

​

*# Show*

t[**-**1]

Out[11]:

2

*# Use indexing just like we did in lists*

t[0:3:1]

Out[15]:

'one'

*# Slicing just like a list*

t[**-**1]

Out[17]:

2

**Basic Tuple Methods**

Tuples have built-in methods, but not as many as lists do. Let's look at two of them:

*# Use .index to enter a value and return the index*

​

t.index('one')

print(dir(t))

Out[20]:

0

*# Use .count to count the number of times a value appears*

t.count('one')

Out[21]:

2

**Immutability**

It can't be stressed enough that tuples are immutable. To drive that point home:

t[0]**=** 'change' *#Can not modify*

**---------------------------------------------------------------------------**

**TypeError** Traceback (most recent call last)

**<ipython-input-22-1257c0aa9edd>** in <module>

**----> 1** t**[0]=** **'change'**

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

Because of this immutability, tuples can't grow. Once a tuple is made we can not add to it.

t.append('nope') *#Can not append as can not be modified*

**---------------------------------------------------------------------------**

**AttributeError** Traceback (most recent call last)

**<ipython-input-23-b75f5b09ac19>** in <module>

**----> 1** t**.**append**('nope')**

**AttributeError**: 'tuple' object has no attribute 'append'

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

*# new tuple will be created instead of updating the exisiting the tuple , while showing the example add*

*# one more value to the tuple*

​

t **=** ('A','B') **+** t[1:]

*#print(t1)*

print(t)

('A', 'B', 's', 'h', 'a')

*#few more ways of adding the values to Tuple*

tup1 **=** () *#empty tuple*

tup1 **=**(0,1,2,3)

print(id(tup1))*#if needed can fill the tuple with values*

tup1 **=** tup1 **+** tuple(range(4,11)) *#can replace one tuple with other , the original tuple will not get modified*

​

print(tup1)

​

109338816

(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

*# tup2*

print(id(tup1))

*# print(id(tup2))*

109383872

tup1 **=** (2,3,4,[5,6]) *#tuple can consist of mutable objects , and these can be modified.*

print(tup1[3][0])

​

tup1[3][0]**=** 3

print(tup1)

​

*# print(tup1[3][0])*

*# # print(tup1)*

​

*# # # print(tup1[3][0])*

*# tup1[3][0]=8*

*# print(tup1)*

​

5

(2, 3, 4, [3, 6])

​

*# tup1[3] = 7 # can not modify the 3rd object because it is tuple object , will not work*

tup1[3][0] **=** 7 *# but we can modify the contents of list object which is present inside the tuple , will work*

*# tup1[3].append(10) # you can append the values to an mutable object but not to immutable*

*# tup1[1] =12 #error*

tup1

Out[11]:

(2, 3, 4, [7, 6])

*#del tup1[1] #can not delete immutable values*

**del** tup1[3][0] *# can delete mutable values*

tup1

Out[19]:

(2, 3, 4, [6, 10, 10])

**del** tup1 *# but can delete whole tuple*

tup3 **=** (6,1,5)

tup2 **=**(3,1,7,4)

tup4 **=** sorted(tup3,reverse **=** **True**)

*#sorted is a function , if decresing then use reverse = True else by default is will be increasing*

print(tup4)

print(tup3) *#sort will be creating new tuple and updating the values and the original values will be unchanged*

*# tup2.sort() #no sort method*

*# tup2*

print(type(tup4))

​

*# lst = [3,2,5]*

*# lst1 = lst.sort() #will not return any value instead it will update on the original value only*

*# print(lst)*

*# print(lst1)*

[6, 5, 1]

(6, 1, 5)

<class 'list'>

min(tup2)

​

Out[20]:

1

max(tup2)

Out[21]:

5

Tuple assignment One of the unique syntactic features of the Python language is the ability to have a tuple on the left hand side of an assignment statement. This allows you to assign more than one variable at a time when the left hand 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' ) *#packing , packing multiple values into single value*

​

x, y **=** m *#unpacking multiples in to multiple variables*

​

​

x

Out[86]:

'have'

y

Out[87]:

'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

Out[89]:

'have'

y

Out[90]:

'fun'

*#Stylistically when we use a tuple on the left hand side of the assignment statement, we omit the parentheses,*

*#but the following is an equally valid syntax:*

​

m **=** ( 'have', 'fun' )

​

(x, y) **=** m

print(x,y)

​

have 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 have to be the same:

Dictionaries and tuples Dictionaries have a method called items that returns a list of tuples, where each tuple is a key-value pair As you should expect from a dictionary, the items are in no particular order.

d **=** {'a':10, 'b':1, 'c':22}

t **=** d.items()

print(t)

dict\_items([('a', 10), ('b', 1), ('c', 22)])

*#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 **=** {'d':10, 'a':1, 'c':22}

​

t **=** d.items()

t

Out[109]:

dict\_items([('d', 10), ('a', 1), ('c', 22)])

tup1**=** (sorted(t)) *#original value will be unchanged*

print(tup1)

print(t)

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

[('a', 1), ('c', 22), ('d', 10)]

dict\_items([('d', 10), ('a', 1), ('c', 22)])

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:

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).

**for** key, val **in** d.items():

print(val, key)

10 d

1 a

22 c

Again 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

​

Out[129]:

[(10, 'a'), (1, 'b'), (22, 'c')]

*#By hand-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.*

​

​

l.sort(reverse**=True**)

l

Out[130]:

[(22, 'c'), (10, 'a'), (1, 'b')]

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:

​

*#Creating composite key in a dictionary using tuple*

number1 **=** 99999

number2 **=** 11111

name:str **=** "Asha"

lastname:str **=**"Talari"

directory **=** {}

directory[name] **=** number1

directory[name,lastname]**=** number2

**for** keys , values **in** directory.items():

print(keys , values)

Asha 99999

('Asha', 'Talari') 11111