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. This chapter presents one more built-in type, the tuple, and then shows how lists, dictionaries, and tuples work together. You'll have an intuition of how to use tuples based on what you've learned about lists. We can treat them very similarly with the major distinction being that tuples are immutable.

Tuples are immutable

A tuple is a sequence of values. The values can be any type, and they are indexed by integers, so in that respect tuples are a lot like lists. The important difference is that tuples are immutable. Syntactically, a tuple is a comma-separated list of values:

In [1]:
```
t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses:

In [2]:
```
t = ('a', 'b', 'c', 'd', 'e')
```

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

In [3]:
```
t1 = 'a',
type(t1)
```

Out[3]:
```
tuple
```

A value in parentheses is not a tuple:

In [4]:
```
t2 = ('a')
type(t2)
```

Out[4]:
```
str
```

In [5]:
```
# Create a tuple
t = (1,2,3)
```

In [6]:
```
# Check len just like a list
len(t)
```

Out[6]: 3

In [7]:
```
# Can also mix object types
t = ('one',2)

# Show
t
```

Out[7]: ('one', 2)

In [8]:
```
# Use indexing just like we did in lists. The bracket operator indexes an
t[0]
```

Out[8]: 'one'

In [9]:
```
# Slicing just like a list
t[-1]
```

Out[9]: 2

And the slice operator selects a range of elements.

In [10]:
```
t = (1,2,3,4)
t[1:3]
```

Out[10]: (2, 3)

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

In [11]:
```
t = tuple()
t
```

Out[11]: ()

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

In [12]:
```
t = tuple('lupins')
t
```

Out[12]: ('l', 'u', 'p', 'i', 'n', 's')

Because tuple is the name of a built-in function, you should avoid using it as a variable name. Most list operators also work on tuples. But if you try to modify one of the elements of the tuple, you get an error:

In [13]:
```
t[0] = 'A'
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
Cell In[13], line 1
----> 1 t[0] = 'A'

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

Because tuples are immutable, you can't modify the elements. But you can replace one tuple with another:

In [14]:
```
t = ('A',) + t[1:]
t
```

Out[14]:
```
('A', 'u', 'p', 'i', 'n', 's')
```

This statement makes a new tuple and then makes t refer to it. The relational 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 elements, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

In [15]:
```
(0, 1, 2) < (0, 3, 4)
```

Out[15]:
```
True
```

In [16]:
```
(0, 1, 2000000) < (0, 3, 4)
```

Out[16]:
```
True
```

Tuple assignment

It is often useful to swap the values of two variables. With conventional assignments, you have to use a temporary variable. For example, to swap a and b:

In [17]:
```
a=2
b=3
temp = a
a = b
b = temp
```

In [18]:
```
a
```

Out[18]:
```
3
```

In [19]:
```
b
```

Out[19]:
```
2
```

This solution is cumbersome; tuple assignment is more elegant:

In [20]:
```
a, b = b, a
```

In [21]:
```
a
```

Out[21]:
```
2
```

In [22]:
```
b
```

Out[22]:
```
3
```

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable. 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:

In [23]:
```
a, b = 1, 2, 3
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
Cell In[23], line 1
----> 1 a, b = 1, 2, 3

ValueError: too many values to unpack (expected 2)
```

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:

In [24]:
```
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.

In [25]:
```
uname
```

Out[25]:
```
'monty'
```

In [26]:
```
domain
```

Out[26]:
```
'python.org'
```

Tuples as return values

Strictly speaking, a function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if you want to divide two integers and compute the quotient and remainder, it is inefficient to compute x//y and then x%y. It is better to compute them both at

the same time. The built-in function divmod takes two arguments and returns a tuple of two values, the quotient and remainder. You can store the result as a tuple:

In [27]:
```
t = divmod(7, 3)
t
```

Out[27]: (2, 1)

Or use tuple assignment to store the elements separately:

In [28]:
```
quot, rem = divmod(7, 3)
quot
```

Out[28]: 2

In [29]:
```
rem
```

Out[29]: 1

Here is an example of a function that returns a tuple:

In [30]:
```
def min_max(t):
    return min(t), max(t)
```

max and min are built-in functions that find the largest and smallest elements of a sequence. min_max computes both and returns a tuple of two values.

Variable-length argument tuples

Functions can take a variable number of arguments. A parameter name that begins with * gathers arguments into a tuple. For example, printall takes any number of arguments and prints them:

In [31]:
```
def printall(*args):
    print(args)
```

The gather parameter can have any name you like, but args is conventional. Here's how the function works:

In [32]:
```
printall(1, 2.0, '3')
```

```
(1, 2.0, '3')
```

The complement of gather is scatter. If you have a sequence of values and you want to pass it to a function as multiple arguments, you can use the * operator. For example, divmod takes exactly two arguments; it doesn't work with a tuple:

In [33]:
```
t = (7, 3)
divmod(t)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[33], line 2
      1 t = (7, 3)
----> 2 divmod(t)

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

But if you scatter the tuple, it works:

In [34]:
```
>>> divmod(*t)
```

Out[34]: (2, 1)

Many of the built-in functions use variable-length argument tuples. For example, max and min can take any number of arguments:

In [35]:
```
max(1, 2, 3)
```

Out[35]: 3

But sum does not.

In [36]:
```
sum(1, 2, 3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[36], line 1
----> 1 sum(1, 2, 3)

TypeError: sum() takes at most 2 arguments (3 given)
```

As an exercise, write a function called sum_all that takes any number of arguments and returns their sum.

12.5 Lists and tuples

zip is a built-in function that takes two or more sequences and interleaves them. The name of the function refers to a zipper, which interleaves two rows of teeth. This example zips a string and a list:

In [37]:
```
s = 'abc'
t = [0, 1, 2]
zip(s, t)
```

Out[37]: <zip at 0x714ba17e4840>

The result is a zip object that knows how to iterate through the pairs. The most common use of zip is in a for loop:

In [38]:
```
for pair in zip(s, t):
    print(pair)
```

```
('a', 0)
('b', 1)
('c', 2)
```

A zip object is a kind of iterator, which is any object that iterates through a sequence. Iterators are similar to lists in some ways, but unlike lists, you can't use an index to select an element from an iterator. If you want to use list operators and methods, you can use a zip object to make a list:

In [39]:
```
list(zip(s, t))
```

Out[39]:
```
[('a', 0), ('b', 1), ('c', 2)]
```

The result is a list of tuples; in this example, each tuple contains a character from the string and the corresponding element from the list. If the sequences are not the same length, the result has the length of the shorter one.

In [40]:
```
list(zip('Anne', 'Elk'))
```

Out[40]:
```
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

You can use tuple assignment in a for loop to traverse a list of tuples:

In [41]:
```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print(number, letter)
```

```
0 a
1 b
2 c
```

Each time through the loop, Python selects the next tuple in the list and assigns the elements to letter and number.

If you combine zip, for and tuple assignment, you get a useful idiom for traversing two (or more) sequences at the same time. For example, has_match takes two sequences, t1 and t2, and returns True if there is an index i such that t1[i] == t2[i]:

In [42]:
```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
        return False
```

If you need to traverse the elements of a sequence and their indices, you can use the built-in function enumerate:

In [43]:
```
for index, element in enumerate('abc'):
    print(index, element)
```

```
0 a
1 b
2 c
```

The result from enumerate is an enumerate object, which iterates a sequence of pairs; each pair contains an index (starting from 0) and an element from the given sequence.

Dictionaries and tuples

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

In [44]:
```
d = {'a':0, 'b':1, 'c':2}
t = d.items()
t
```

Out[44]:
```
dict_items([('a', 0), ('b', 1), ('c', 2)])
```

The result is a dict_items object, which is an iterator that iterates the key-value pairs. You can use it in a for loop like this:

In [45]:
```
for key, value in d.items():
    print(key, value)
```

```
a 0
b 1
c 2
```

As you should expect from a dictionary, the items are in no particular order. Going in the other direction, you can use a list of tuples to initialize a new dictionary:

In [46]:
```
t = [('a', 0), ('c', 2), ('b', 1)]
d = dict(t)
d
```

Out[46]:
```
{'a': 0, 'c': 2, 'b': 1}
```

Combining dict with zip yields a concise way to create a dictionary:

In [47]:
```
d = dict(zip('abc', range(3)))
d
```

Out[47]:
```
{'a': 0, 'b': 1, 'c': 2}
```

Sequences of sequences

We 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 should 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: than a list.

1. In some contexts, like a return statement, it is syntactically simpler to create a tuple

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. But Python provides the built-in function sorted, which takes any sequence and returns a new list with the same elements in sorted order, and reversed, which takes a sequence and returns an iterator that traverses the list in reverse order.

Basic Tuple Methods

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

In [48]:
```
# Use .index to enter a value and return the index
t = ('one', 'two', 'three', 'four', 'five')
t.index('one')
```

Out[48]:
```
0
```

In [49]:
```
# Use .count to count the number of times a value appears
t.count('one')
```

Out[49]:
```
1
```

Immutability

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

In [50]:
```
t[0]= 'change'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[50], line 1
----> 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.

In [51]:
```
t.append('nope')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[51], line 1
----> 1 t.append('nope')

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

When to use Tuples

You may be wondering, "Why bother using tuples when they have fewer available methods?" To be honest, tuples are not used as often as lists in programming, but are used when immutability is necessary. If in your program you are passing around an object and need to make sure it does not get changed, then a tuple becomes your solution. It provides a convenient source of data integrity.

You should now be able to create and use tuples in your programming as well as have an understanding of their immutability.

Sets

Sets are an unordered collection of *unique* elements. We can construct them by using the set() function. Let's go ahead and make a set to see how it works

In [52]:
```
x = set()
```

In [53]:
```
# We add to sets with the add() method
x.add(1)
```

In [54]:
```
#Show
x
```

Out[54]:
```
{1}
```

Note the curly brackets. This does not indicate a dictionary! Although you can draw analogies as a set being a dictionary with only keys.

We know that a set has only unique entries. So what happens when we try to add something that is already in a set?

In [55]:
```
# Add a different element
x.add(2)
```

In [56]:
```
#Show
x
```

Out[56]:
```
{1, 2}
```

In [57]:
```
# Try to add the same element
x.add(1)
```

In [58]:
```
#Show
x
```

Out[58]:
```
{1, 2}
```

Notice how it won't place another 1 there. That's because a set is only concerned with unique elements! We can cast a list with multiple repeat elements to a set to get the unique elements. For example:

In [59]:
```
# Create a list with repeats
list1 = [1,1,2,2,3,4,5,6,1,1]
```

In [60]:
```
# Cast as set to get unique values
set(list1)
```

Out[60]:
```
{1, 2, 3, 4, 5, 6}
```