Tuples

WE'LL COVER THE FOLLOWING

A tuple is an immutable list. A tuple can not be changed in any way once it is created.

```
a_tuple = ("a", "b", "mpilgrim", "z", "example")          #①
print (a_tuple)
#('a', 'b', 'mpilgrim', 'z', 'example')

print (a_tuple[0] )                                       #②
#a

print (a_tuple[-1])                                       #③
#example

print (a_tuple[1:3] )                                     #④
#('b', 'mpilgrim')
```

① A tuple is defined in the same way as a list, except that the whole set of elements is enclosed in parentheses instead of square brackets.

② The elements of a tuple have a defined order, just like a list. Tuple indices are zero-based, just like a list, so the first element of a non-empty tuple is always `a_tuple[0]`.

③ Negative indices count from the end of the tuple, just like a list.

④ Slicing works too, just like a list. When you slice a list, you get a new list; when you slice a tuple, you get a new tuple.

The major difference between tuples and lists is that tuples can not be changed. In technical terms, tuples are immutable. In practical terms, they have no methods that would allow you to change them. Lists have methods like `append()`, `extend()`, `insert()`, `remove()`, and `pop()`. Tuples have none of these methods. You can slice a tuple (because that creates a new tuple), and you can check whether a tuple contains a particular value (because that doesn't change the tuple), and... that's about it.

```
# continued from the previous example
a_tuple = ("a", "b", "mpilgrim", "z", "example")
print (a_tuple)
#('a', 'b', 'mpilgrim', 'z', 'example')

print (a_tuple.append("new"))               #①
#Traceback (most recent call last):
# File "/usercode/__ed_file.py", line 6, in <module>
# print (a_tuple.append("new")) #\u2460
#AttributeError: 'tuple' object has no attribute 'append'
```

```
a_tuple = ("a", "b", "mpilgrim", "z", "example")
print (a_tuple.remove("z"))                 #②
#Traceback (most recent call last):
# File "/usercode/__ed_file.py", line 2, in <module>
# print (a_tuple.remove("z")) #\u2461
#AttributeError: 'tuple' object has no attribute 'remove
```

```
a_tuple = ("a", "b", "mpilgrim", "z", "example")

print (a_tuple.index("example") )           #③
#4

print ("z" in a_tuple)                      #④
#True
```

① You can't add elements to a tuple. Tuples have no `append()` or `extend()` method.

② You can't remove elements from a tuple. Tuples have no `remove()` or `pop()` method.

③ You *can* find elements in a tuple, since this doesn't change the tuple.

④ You can also use the in operator to check if an element exists in the tuple.

So what are tuples good for?

1. Tuples are faster than lists. If you're defining a constant set of values and all you're ever going to do with it is iterate through it, use a tuple instead of a list.
2. It makes your code safer if you "write-protect" data that doesn't need to be changed. Using a tuple instead of a list is like having an implied `assert` statement that shows this data is constant, and that special thought (and a specific function) is required to override that.
3. Some tuples can be used as dictionary keys (specifically, tuples that contain *immutable* values like strings, numbers, and other tuples). Lists can never be used as dictionary keys, because lists are not immutable.

> *Tuples can be converted into lists, and vice-versa. The built-in **tuple()** function takes a list and returns a tuple with the same elements, and the **list()** function takes a tuple and returns a list. In effect, **tuple()** freezes a list, and **list()** thaws a tuple.*

Tuples In A Boolean Context

You can use tuples in a boolean context, such as an `if` statement.

```
def is_it_true(anything):
   if anything:
     print("yes, it's true")
   else:
     print("no, it's false")

print (is_it_true(()))             #①
#no, it's false
#None

print (is_it_true(('a', 'b')))     #②
```

```
#yes, it's true
#None


print (is_it_true((False,)))       #③
#yes, it's true
#None

print (type((False)))              #④
#<class 'bool'>

print (type((False,)))
#<class 'tuple'>
```

① In a boolean context, an empty tuple is false.

② Any tuple with at least one item is true.

③ Any tuple with at least one item is true. The value of the items is irrelevant. But what's that comma doing there?

④ To create a tuple of one item, you need a comma after the value. Without the comma, Python just assumes you have an extra pair of parentheses, which is harmless, but it doesn't create a tuple.

Assigning Multiple Values At Once

Here's a cool programming shortcut: in Python, you can use a tuple to assign multiple values at once.

```
v = ('a', 2, True)
(x, y, z) = v       #①
print (x)
#a

print (y)
#2

print (z)
#True
```

① `v` is a tuple of three elements, and `(x, y, z)` is a tuple of three variables. Assigning one to the other assigns each of the values of v to each of the

variables, in order.

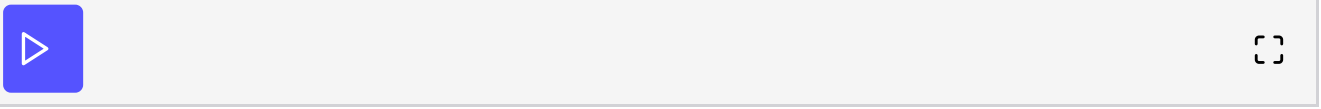
This has all kinds of uses. Suppose you want to assign names to a range of values. You can use the built-in `range()` function with multi-variable assignment to quickly assign consecutive values.

```
(MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY) = range(7)  #①

print (MONDAY)                                                                #②
#0

print (TUESDAY)
#1

print (SUNDAY)
#6
```



① The built-in `range()` function constructs a sequence of integers. (Technically, the `range()` function returns an iterator, not a list or a tuple, but you'll learn about that distinction later.) `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`, and `SUNDAY` are the variables you're defining. (This example came from the `calendar` module, a fun little module that prints calendars, like the `UNIX` program `cal`. The `calendar` module defines integer constants for days of the week.)

② Now each variable has its value: `MONDAY` is 0, `TUESDAY` is 1, and so forth.

You can also use multi-variable assignment to build functions that return multiple values, simply by returning a tuple of all the values. The caller can treat it as a single tuple, or it can assign the values to individual variables. Many standard Python libraries do this, including the os module, which you'll learn about in the next chapter.