Lecture 3-3

Tuples

Week 3 Friday

Miles Chen, PhD

Adapted from Chapter 12 of Think Python by Allen B Downey

Additional content on Dictionaries adapted from "Whirlwind Tour of Python" by Jake VanderPlas

Tuples

Tuples are like lists in that they can contain objects of different types. Tuples are ordered and preserve their order.

Tuples are different from lists in that they are **immutable**. You cannot append to a tuple or modify values in a tuple.

Tuples are created if you write values separated by commas. You can also use curved brackets (parenthesis) `()`.

Tuples

Tuples are like lists in that they can contain objects of different types. Tuples are ordered and preserve their order.

Tuples are different from lists in that they are **immutable**. You cannot append to a tuple or modify values in a tuple.

Tuples are created if you write values separated by commas. You can also use curved brackets (parenthesis) `()`.

In [1]:
```
t = 0, 'apple', 2, 'cat', 'dog', 5, 6  # parentheses are not required to create a tuple
```

Tuples

Tuples are like lists in that they can contain objects of different types. Tuples are ordered and preserve their order.

Tuples are different from lists in that they are **immutable**. You cannot append to a tuple or modify values in a tuple.

Tuples are created if you write values separated by commas. You can also use curved brackets (parenthesis) `()`.

In [1]:
```
t = 0, 'apple', 2, 'cat', 'dog', 5, 6  # parentheses are not required to create a tuple
```

In [2]:
```
t
```

Out[2]:
```
(0, 'apple', 2, 'cat', 'dog', 5, 6)
```

Tuples

Tuples are like lists in that they can contain objects of different types. Tuples are ordered and preserve their order.

Tuples are different from lists in that they are **immutable**. You cannot append to a tuple or modify values in a tuple.

Tuples are created if you write values separated by commas. You can also use curved brackets (parenthesis) `()`.

In [1]:
```
t = 0, 'apple', 2, 'cat', 'dog', 5, 6  # parentheses are not required to create a tuple
```

In [2]:
```
t
```

Out[2]:
```
(0, 'apple', 2, 'cat', 'dog', 5, 6)
```

In [3]:
```
t = (0, 'apple', 2, 'cat', 'dog', 5, 6) # you can use parentheses
```

Tuples

Tuples are like lists in that they can contain objects of different types. Tuples are ordered and preserve their order.

Tuples are different from lists in that they are **immutable**. You cannot append to a tuple or modify values in a tuple.

Tuples are created if you write values separated by commas. You can also use curved brackets (parenthesis) `()`.

In [1]:
```
t = 0, 'apple', 2, 'cat', 'dog', 5, 6  # parentheses are not required to create a tuple
```

In [2]:
```
t
```

Out[2]:
```
(0, 'apple', 2, 'cat', 'dog', 5, 6)
```

In [3]:
```
t = (0, 'apple', 2, 'cat', 'dog', 5, 6) # you can use parentheses
```

In [4]:
```
t
```

Out[4]:
```
(0, 'apple', 2, 'cat', 'dog', 5, 6)
```

You can create a tuple with one value by having a trailing comma

You can create a tuple with one value by having a trailing comma

In [5]:
```
t1 = "a",
```

You can create a tuple with one value by having a trailing comma

In [5]:
```
t1 = "a",
```

In [6]:
```
t1
```

Out[6]:
```
('a',)
```

You can create a tuple with one value by having a trailing comma

In [5]:
```
t1 = "a",
```

In [6]:
```
t1
```

Out[6]:
```
('a',)
```

In [7]:
```
type(t1)
```

Out[7]:
```
tuple
```

You can create a tuple with one value by having a trailing comma

In [5]:
```
t1 = "a",
```

In [6]:
```
t1
```

Out[6]:
```
('a',)
```

In [7]:
```
type(t1)
```

Out[7]:
```
tuple
```

In [8]:
```
len(t1)
```

Out[8]:
```
1
```

using parentheses without a comma will not work

using parentheses without a comma will not work

In [9]:
```
t2 = ("a")
```

using parentheses without a comma will not work

In [9]:
```
t2 = ("a")
```

In [10]:
```
type(t2)
```

Out[10]:
```
str
```

using parentheses without a comma will not work

In [9]:
```
t2 = ("a")
```

In [10]:
```
type(t2)
```

Out[10]:
```
str
```

In [11]:
```
t2 = ("a",)
```

using parentheses without a comma will not work

In [9]:
```
t2 = ("a")
```

In [10]:
```
type(t2)
```

Out[10]:
```
str
```

In [11]:
```
t2 = ("a",)
```

In [12]:
```
type(t2)
```

Out[12]:
```
tuple
```

You can create an empty tuple with the `tuple()` function, similar to using the `list()` or `dict()` function

You can create an empty tuple with the `tuple()` function, similar to using the `list()` or `dict()` function

In [13]:
```
t3 = tuple()
```

You can create an empty tuple with the `tuple()` function, similar to using the `list()` or `dict()` function

In [13]:
```
t3 = tuple()
```

In [14]:
```
t3
```

Out[14]:
```
()
```

You can use the `tuple()` function to turn other iterables into tuples.

You can use the `tuple()` function to turn other iterables into tuples.

In [15]:
```
tuple("hello")
```

Out[15]:
```
('h', 'e', 'l', 'l', 'o')
```

You can use the `tuple()` function to turn other iterables into tuples.

In [15]:
```
tuple("hello")
```

Out[15]:
```
('h', 'e', 'l', 'l', 'o')
```

In [16]:
```
tuple(range(5))
```

Out[16]:
```
(0, 1, 2, 3, 4)
```

You can use the `tuple()` function to turn other iterables into tuples.

In [15]:
```
tuple("hello")
```

Out[15]:
```
('h', 'e', 'l', 'l', 'o')
```

In [16]:
```
tuple(range(5))
```

Out[16]:
```
(0, 1, 2, 3, 4)
```

In [17]:
```
tuple([1,4,7])
```

Out[17]:
```
(1, 4, 7)
```

The usual indexing rules apply to tuples

The usual indexing rules apply to tuples

In [18]:
```
t = 0, 'apple', 2, 'cat', 'dog', 5, 6
```

The usual indexing rules apply to tuples

In [18]:
```
t = 0, 'apple', 2, 'cat', 'dog', 5, 6
```

In [19]:
```
t[1]
```

Out[19]:
```
'apple'
```

The usual indexing rules apply to tuples

In [18]:
```
t = 0, 'apple', 2, 'cat', 'dog', 5, 6
```

In [19]:
```
t[1]
```

Out[19]:
```
'apple'
```

In [20]:
```
t[2:5] # slicing
```

Out[20]:
```
(2, 'cat', 'dog')
```

Tuples are immutable. They cannot be modified.

List are mutable and can be modified

Tuples are immutable. They cannot be modified.

List are mutable and can be modified

In [21]:

```
t = (0,'apple',2,'cat','dog',5,6) # tuple
l = [0,'apple',2,'cat','dog',5,6] # list
```

Tuples are immutable. They cannot be modified.

List are mutable and can be modified

In [21]:
```
t = (0,'apple',2,'cat','dog',5,6) # tuple
l = [0,'apple',2,'cat','dog',5,6] # list
```

In [22]:
```
l[0] = 100  # we can change the value of the object at index 0
print(l)
```

```
[100, 'apple', 2, 'cat', 'dog', 5, 6]
```

Tuples are immutable. They cannot be modified.

List are mutable and can be modified

In [21]:
```
t = (0,'apple',2,'cat','dog',5,6) # tuple
l = [0,'apple',2,'cat','dog',5,6] # list
```

In [22]:
```
l[0] = 100  # we can change the value of the object at index 0
print(l)
```

```
[100, 'apple', 2, 'cat', 'dog', 5, 6]
```

In [23]:
```
t[0] = 100  # trying to modify the value in a tuple is not allowed
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-23-1315e91aabf3> in <module>
----> 1 t[0] = 100  # trying to modify the value in a tuple is not allowed

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

methods that modify lists in place (e.g. append, insert, pop, etc) do not work for tuples

methods that modify lists in place (e.g. append, insert, pop, etc) do not work for tuples

In [24]:
```
l.append('x')
print(l)
```

```
[100, 'apple', 2, 'cat', 'dog', 5, 6, 'x']
```

methods that modify lists in place (e.g. append, insert, pop, etc) do not work for tuples

In [24]:
```
l.append('x')
print(l)
```

```
[100, 'apple', 2, 'cat', 'dog', 5, 6, 'x']
```

In [25]:
```
t.append('x')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-25-2bc04b290b67> in <module>
----> 1 t.append('x')

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

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

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

In [26]:
```
t = ("A",) + t[1:]
t
```

Out[26]:
```
('A', 'apple', 2, 'cat', 'dog', 5, 6)
```

This creates an entirely new tuple, unrelated to the other one.

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).

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 [27]:
```
(0, 1, 2) < (0, 3, 4)
```

Out[27]:
```
True
```

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 [27]:
```
(0, 1, 2) < (0, 3, 4)
```

Out[27]:
```
True
```

In [28]:
```
(0, 1, 2000000) < (0, 3, 4)
```

Out[28]:
```
True
```

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 [27]:
```
(0, 1, 2) < (0, 3, 4)
```

Out[27]:
```
True
```

In [28]:
```
(0, 1, 2000000) < (0, 3, 4)
```

Out[28]:
```
True
```

In [29]:
```
(0, 500, 2) < (0, 3, 4)
```

Out[29]:
```
False
```

Tuple assignment

A common and useful tuple idiom: You can switch value assignments via tuples

Tuple assignment

A common and useful tuple idiom: You can switch value assignments via tuples

In [30]:
```
# old option without tuples
a = 5
b = 1

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

```
1 5
```

Tuple assignment

A common and useful tuple idiom: You can switch value assignments via tuples

In [30]:
```
# old option without tuples
a = 5
b = 1

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

```
1 5
```

In [31]:
```
# faster way with tuples
a = 5
b = 1

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

```
1 5
```

You can take the results of a function and have the returned values assign to different elements in a tuple

You can take the results of a function and have the returned values assign to different elements in a tuple

In [32]:

```
addr = "mileschen@stat.ucla.edu"
uname, domain = addr.split("@")
```

You can take the results of a function and have the returned values assign to different elements in a tuple

In [32]:
```
addr = "mileschen@stat.ucla.edu"
uname, domain = addr.split("@")
```

In [33]:
```
uname
```

Out[33]:
```
'mileschen'
```

You can take the results of a function and have the returned values assign to different elements in a tuple

In [32]:
```
addr = "mileschen@stat.ucla.edu"
uname, domain = addr.split("@")
```

In [33]:
```
uname
```

Out[33]:
```
'mileschen'
```

In [34]:
```
domain
```

Out[34]:
```
'stat.ucla.edu'
```

We saw this when we talked about functions. You can have functions return multiple values in the form of a tuple

We saw this when we talked about functions. You can have functions return multiple values in the form of a tuple

In [35]:
```
def my_divide(x, y):
    integer = x // y
    remainder = x % y
    return integer, remainder
```

We saw this when we talked about functions. You can have functions return multiple values in the form of a tuple

In [35]:
```
def my_divide(x, y):
    integer = x // y
    remainder = x % y
    return integer, remainder
```

In [36]:
```
a, b = my_divide(23, 5)
```

We saw this when we talked about functions. You can have functions return multiple values in the form of a tuple

In [35]:
```
def my_divide(x, y):
    integer = x // y
    remainder = x % y
    return integer, remainder
```

In [36]:
```
a, b = my_divide(23, 5)
```

In [37]:
```
a, b
```

Out[37]:
```
(4, 3)
```

We saw this when we talked about functions. You can have functions return multiple values in the form of a tuple

In [35]:
```
def my_divide(x, y):
    integer = x // y
    remainder = x % y
    return integer, remainder
```

In [36]:
```
a, b = my_divide(23, 5)
```

In [37]:
```
a, b
```

Out[37]:
```
(4, 3)
```

In [38]:
```
divmod(23, 5)  # divmod() is a built-in function that does exactly this.
```

Out[38]:
```
(4, 3)
```

Tuple Methods

tuples only support two methods: `tuple.index()` and `tuple.count()` which return information about contents of the tuple but do not modify them

Tuple Methods

tuples only support two methods: `tuple.index()` and `tuple.count()` which return information about contents of the tuple but do not modify them

In [39]:
```
t = 0, 'apple', 2, 'cat', 'dog', 5, 6
```

These methods also work for lists

Tuple Methods

tuples only support two methods: `tuple.index()` and `tuple.count()` which return information about contents of the tuple but do not modify them

In [39]:
```
t = 0, 'apple', 2, 'cat', 'dog', 5, 6
```

In [40]:
```
t.index('dog')
```

Out[40]:
```
4
```

Tuple Methods

tuples only support two methods: `tuple.index()` and `tuple.count()` which return information about contents of the tuple but do not modify them

In [39]:
```
t = 0, 'apple', 2, 'cat', 'dog', 5, 6
```

In [40]:
```
t.index('dog')
```

Out[40]:
```
4
```

In [41]:
```
t.count(5)
```

Out[41]:
```
1
```

Functions that support tuples and other iterables as inputs

Even though tuples only have two methods, there are several functions that support tuples (and other iterables like lists, dicts, strings) as inputs

- `len()`
- `sum()`
- `sorted()`
- `min()`
- `max()`

None of these functions affect the list or tuple itself.

In [42]:

```
some_digits = (4,2,7,9,2,5,3)  # a tuple of numbers
some_words = ['dog','apple','cat','hat','hand']  # this is a list
```

In [42]:
```
some_digits = (4,2,7,9,2,5,3)  # a tuple of numbers
some_words = ['dog','apple','cat','hat','hand']  # this is a list
```

In [43]:
```
len(some_digits)
```

Out[43]:
```
7
```

In [42]:
```
some_digits = (4,2,7,9,2,5,3)  # a tuple of numbers
some_words = ['dog','apple','cat','hat','hand']  # this is a list
```

In [43]:
```
len(some_digits)
```

Out[43]: 7

In [44]:
```
sum(some_digits)
```

Out[44]: 32

In [42]:
```
some_digits = (4,2,7,9,2,5,3)  # a tuple of numbers
some_words = ['dog','apple','cat','hat','hand']  # this is a list
```

In [43]:
```
len(some_digits)
```

Out[43]:
```
7
```

In [44]:
```
sum(some_digits)
```

Out[44]:
```
32
```

In [45]:
```
sum(some_words) # won't work on strings
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-45-7ad2d781cfbf> in <module>
----> 1 sum(some_words) # won't work on strings

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In [46]:

```
sorted(some_digits)  # sorts the tuple, but does not affect the list or tuple itself.
# contrast to list.sort() which will sort the list in place
# but the object returned is a list
```

Out[46]:

```
[2, 2, 3, 4, 5, 7, 9]
```

In [46]:
```
sorted(some_digits)  # sorts the tuple, but does not affect the list or tuple itself.
# contrast to list.sort() which will sort the list in place
# but the object returned is a list
```

Out[46]:
```
[2, 2, 3, 4, 5, 7, 9]
```

In [47]:
```
print(some_digits)  # just to show the list is unchanged
```

```
(4, 2, 7, 9, 2, 5, 3)
```

In [46]:
```
sorted(some_digits)  # sorts the tuple, but does not affect the list or tuple itself.
# contrast to list.sort() which will sort the list in place
# but the object returned is a list
```

Out[46]:
```
[2, 2, 3, 4, 5, 7, 9]
```

In [47]:
```
print(some_digits)  # just to show the list is unchanged
```

```
(4, 2, 7, 9, 2, 5, 3)
```

In [48]:
```
sorted(some_words) # when applied to a list of strings, it will alphabetize them
```

Out[48]:
```
['apple', 'cat', 'dog', 'hand', 'hat']
```

In [46]:
```
sorted(some_digits)  # sorts the tuple, but does not affect the list or tuple itself.
# contrast to list.sort() which will sort the list in place
# but the object returned is a list
```

Out[46]:
```
[2, 2, 3, 4, 5, 7, 9]
```

In [47]:
```
print(some_digits)  # just to show the list is unchanged
```

```
(4, 2, 7, 9, 2, 5, 3)
```

In [48]:
```
sorted(some_words) # when applied to a list of strings, it will alphabetize them
```

Out[48]:
```
['apple', 'cat', 'dog', 'hand', 'hat']
```

In [49]:
```
min(some_digits)
```

Out[49]:
```
2
```

In [46]:
```
sorted(some_digits)  # sorts the tuple, but does not affect the list or tuple itself.
# contrast to list.sort() which will sort the list in place
# but the object returned is a list
```

Out[46]:
```
[2, 2, 3, 4, 5, 7, 9]
```

In [47]:
```
print(some_digits)  # just to show the list is unchanged
```

```
(4, 2, 7, 9, 2, 5, 3)
```

In [48]:
```
sorted(some_words) # when applied to a list of strings, it will alphabetize them
```

Out[48]:
```
['apple', 'cat', 'dog', 'hand', 'hat']
```

In [49]:
```
min(some_digits)
```

Out[49]:
```
2
```

In [50]:
```
max(some_words)  # max returns the last word if alphabetized,
# min will return the first in an alphabetized list
```

Out[50]:
```
'hat'
```

In [51]:

```
dict_num = {1: "a", 2: "b", 3: "c"}
dict_alpha = {"a":1, "b":2, "c":3}
```

In [51]:
```
dict_num = {1: "a", 2: "b", 3: "c"}
dict_alpha = {"a":1, "b":2, "c":3}
```

In [52]:
```
len(dict_num) # number of items in the dictionary
```

Out[52]:
```
3
```

In [51]:
```
dict_num = {1: "a", 2: "b", 3: "c"}
dict_alpha = {"a":1, "b":2, "c":3}
```

In [52]:
```
len(dict_num) # number of items in the dictionary
```

Out[52]:
```
3
```

In [53]:
```
sorted(dict_num) # a list of the keys sorted
```

Out[53]:
```
[1, 2, 3]
```

In [51]:
```
dict_num = {1: "a", 2: "b", 3: "c"}
dict_alpha = {"a":1, "b":2, "c":3}
```

In [52]:
```
len(dict_num) # number of items in the dictionary
```

Out[52]:
```
3
```

In [53]:
```
sorted(dict_num) # a list of the keys sorted
```

Out[53]:
```
[1, 2, 3]
```

In [54]:
```
sorted(dict_alpha)
```

Out[54]:
```
['a', 'b', 'c']
```

In [51]:
```
dict_num = {1: "a", 2: "b", 3: "c"}
dict_alpha = {"a":1, "b":2, "c":3}
```

In [52]:
```
len(dict_num) # number of items in the dictionary
```

Out[52]:
```
3
```

In [53]:
```
sorted(dict_num) # a list of the keys sorted
```

Out[53]:
```
[1, 2, 3]
```

In [54]:
```
sorted(dict_alpha)
```

Out[54]:
```
['a', 'b', 'c']
```

In [55]:
```
max(dict_num) # the "maximum" key
```

Out[55]:
```
3
```

Math operators and lists, tuples, strings

multiplication generally duplicates

addition generally appends

behaviors across lists, tuples, and strings are similar

Math operators and lists, tuples, strings

multiplication generally duplicates

addition generally appends

behaviors across lists, tuples, and strings are similar

In [56]:

```
L1 = ['a','b','c']
L2 = ['d','e','f']
```

Math operators and lists, tuples, strings

multiplication generally duplicates

addition generally appends

behaviors across lists, tuples, and strings are similar

In [56]:
```
L1 = ['a','b','c']
L2 = ['d','e','f']
```

In [57]:
```
L1 * 2 # multiplication extends duplicates
```

Out[57]:
```
['a', 'b', 'c', 'a', 'b', 'c']
```

Math operators and lists, tuples, strings

multiplication generally duplicates

addition generally appends

behaviors across lists, tuples, and strings are similar

In [56]:
```
L1 = ['a','b','c']
L2 = ['d','e','f']
```

In [57]:
```
L1 * 2 # multiplication extends duplicates
```

Out[57]:
```
['a', 'b', 'c', 'a', 'b', 'c']
```

In [58]:
```
L1 + L2 # addition appends list objects
```

Out[58]:
```
['a', 'b', 'c', 'd', 'e', 'f']
```

In [59]:

```
T1 = ('a','b','c')
T2 = ('d','e','f')
```

In [59]:
```
T1 = ('a','b','c')
T2 = ('d','e','f')
```

In [60]:
```
T1 * 2
```

Out[60]:
```
('a', 'b', 'c', 'a', 'b', 'c')
```

In [61]:
```
T1 + T2
```

Out[61]:
```
('a', 'b', 'c', 'd', 'e', 'f')
```

In [62]:
```
L1 + T2 # fails. you cannot add list and tuple
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-62-75bf80180df8> in <module>
----> 1 L1 + T2 # fails. you cannot add list and tuple

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

In [63]:
```
L1 + list(T2) # but you can easily convert a tuple to a list first
```

Out[63]:
```
['a', 'b', 'c', 'd', 'e', 'f']
```

Variable-length argument tuples

Functions can take a variable number of arguments. A parameter name that begins with `*` gathers arguments into a tuple.

The gather parameter can have any name you like, but args is conventional.

Variable-length argument tuples

Functions can take a variable number of arguments. A parameter name that begins with `*` gathers arguments into a tuple.

The gather parameter can have any name you like, but args is conventional.

In [64]:
```
def printall(*args):
    print(args)
```

Variable-length argument tuples

Functions can take a variable number of arguments. A parameter name that begins with `*` gathers arguments into a tuple.

The gather parameter can have any name you like, but args is conventional.

In [64]:
```
def printall(*args):
    print(args)
```

In [65]:
```
printall(1, 3.0, 5, "hi")
```

```
(1, 3.0, 5, 'hi')
```

In [66]:

```
def print_lines(*args):
    for element in args:
        print(element)
```

In [66]:

```
def print_lines(*args):
    for element in args:
        print(element)
```

In [67]:

```
print_lines("hi", "goodbye")
```

```
hi
goodbye
```

In [66]:
```
def print_lines(*args):
    for element in args:
        print(element)
```

In [67]:
```
print_lines("hi", "goodbye")
```

```
hi
goodbye
```

In [68]:
```
print_lines(1, 5, 7, 9, 10)
```

```
1
5
7
9
10
```

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, the `my_divide` function from earlier takes exactly two arguments; it doesn't work with a tuple:

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, the `my_divide` function from earlier takes exactly two arguments; it doesn't work with a tuple:

In [69]:
```
def my_divide(x, y):
    integer = x // y
    remainder = x % y
    return integer, remainder
```

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, the `my_divide` function from earlier takes exactly two arguments; it doesn't work with a tuple:

In [69]:
```
def my_divide(x, y):
    integer = x // y
    remainder = x % y
    return integer, remainder
```

In [70]:
```
t = (23, 5)
```

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, the `my_divide` function from earlier takes exactly two arguments; it doesn't work with a tuple:

In [69]:
```
def my_divide(x, y):
    integer = x // y
    remainder = x % y
    return integer, remainder
```

In [70]:
```
t = (23, 5)
```

In [71]:
```
my_divide(t)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-71-ea0afdae4ba1> in <module>
----> 1 my_divide(t)

TypeError: my_divide() missing 1 required positional argument: 'y'
```

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, the `my_divide` function from earlier takes exactly two arguments; it doesn't work with a tuple:

In [69]:
```
def my_divide(x, y):
    integer = x // y
    remainder = x % y
    return integer, remainder
```

In [70]:
```
t = (23, 5)
```

In [71]:
```
my_divide(t)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-71-ea0afdae4ba1> in <module>
----> 1 my_divide(t)

TypeError: my_divide() missing 1 required positional argument: 'y'
```

In [72]:
```
my_divide(*t)
```

Out[72]:
```
(4, 3)
```

Lists, Tuples and Iterators

`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.

Lists, Tuples and Iterators

`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.

In [73]:
```
s = 'abc'
t = 0, 1, 2
zip(s, t)
```

Out[73]:
```
<zip at 0x1bbe6b81388>
```

Lists, Tuples and Iterators

`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.

In [73]:
```
s = 'abc'
t = 0, 1, 2
zip(s, t)
```

The zip object creates a sequence of tuples

Out[73]:
```
<zip at 0x1bbe6b81388>
```

In [74]:
```
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, you can put the zip object inside a list.

It will return a list of tuples

If you want, you can put the zip object inside a list.

It will return a list of tuples

In [75]:
```
list(zip(s,t))
```

Out[75]:
```
[('a', 0), ('b', 1), ('c', 2)]
```

If you want, you can put the zip object inside a list.

It will return a list of tuples

In [75]:
```
list(zip(s,t))
```

Out[75]:
```
[('a', 0), ('b', 1), ('c', 2)]
```

If the sequences are not the same length, the result has the length of the shorter one.

If you want, you can put the zip object inside a list.

It will return a list of tuples

In [75]:
```
list(zip(s,t))
```

Out[75]:
```
[('a', 0), ('b', 1), ('c', 2)]
```

If the sequences are not the same length, the result has the length of the shorter one.

In [76]:
```
list(zip("Anne" , "Elk" ))
```

Out[76]:
```
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

If you have a list of tuples, you can iterate over them by unpacking the elements.

If you have a list of tuples, you can iterate over them by unpacking the elements.

In [77]:
```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print(number, letter)
```

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

A useful snippet of code that will traverse two iterables and see if there both share a certain element in the same position.

A useful snippet of code that will traverse two iterables and see if there both share a certain element in the same position.

In [78]:
```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

A useful snippet of code that will traverse two iterables and see if there both share a certain element in the same position.

In [78]:
```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

In [79]:
```
has_match(('a', 'b', 'c'), ('d', 'e', 'f'))
```

Out[79]:
```
False
```

A useful snippet of code that will traverse two iterables and see if there both share a certain element in the same position.

In [78]:
```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

In [79]:
```
has_match(('a', 'b', 'c'), ('d', 'e', 'f'))
```

Out[79]:
```
False
```

In [80]:
```
has_match(('a', 'b', 'c'), ('d', 'e', 'c'))
```

Out[80]:
```
True
```

A useful snippet of code that will traverse two iterables and see if there both share a certain element in the same position.

In [78]:
```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

In [79]:
```
has_match(('a', 'b', 'c'), ('d', 'e', 'f'))
```

Out[79]: False

In [80]:
```
has_match(('a', 'b', 'c'), ('d', 'e', 'c'))
```

Out[80]: True

In [81]:
```
has_match(('a', 'b', 'c', 'd'), ('d', 'c', 'b', 'a'))
```

Out[81]: False

`enumerate()`

The built-in function enumerate is useful. It takes an iterable and returns an iterator of the index paired with the elements You can think of `enumerate()` as zipping a range object of the same length with the iterable.

`enumerate()`

The built-in function enumerate is useful. It takes an iterable and returns an iterator of the index paired with the elements You can think of `enumerate()` as zipping a range object of the same length with the iterable.

In [82]:
```
enumerate("morning")
```

Out[82]:
```
<enumerate at 0x1bbe6b4c598>
```

`enumerate()`

The built-in function enumerate is useful. It takes an iterable and returns an iterator of the index paired with the elements You can think of `enumerate()` as zipping a range object of the same length with the iterable.

In [82]:
```
enumerate("morning")
```

Out[82]:
```
<enumerate at 0x1bbe6b4c598>
```

In [83]:
```
for index, value in enumerate("morning"):
    print(index, value)
```

```
0 m
1 o
2 r
3 n
4 i
5 n
6 g
```

`enumerate()`

The built-in function enumerate is useful. It takes an iterable and returns an iterator of the index paired with the elements You can think of `enumerate()` as zipping a range object of the same length with the iterable.

In [82]:
```
enumerate("morning")
```

Out[82]:
```
<enumerate at 0x1bbe6b4c598>
```

In [83]:
```
for index, value in enumerate("morning"):
    print(index, value)
```

```
0 m
1 o
2 r
3 n
4 i
5 n
6 g
```

In [84]:
```
list(enumerate(['a','b','c','d']))
```

Out[84]:
```
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

`enumerate()`

The built-in function enumerate is useful. It takes an iterable and returns an iterator of the index paired with the elements You can think of `enumerate()` as zipping a range object of the same length with the iterable.

In [82]:
```
enumerate("morning")
```

Out[82]:
```
<enumerate at 0x1bbe6b4c598>
```

In [83]:
```
for index, value in enumerate("morning"):
    print(index, value)
```

```
0 m
1 o
2 r
3 n
4 i
5 n
6 g
```

In [84]:
```
list(enumerate(['a','b','c','d']))
```

Out[84]:
```
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

In [85]:
```
list(zip(range(4),['a','b','c','d']))  # zipping a range object with a list
```

Out[85]:
```
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

Dictionaries and Tuples

the dictionary view object, dict.items() is a sequence of tuples.

Dictionaries and Tuples

the dictionary view object, dict.items() is a sequence of tuples.

In [86]:
```
d = {'a':0, 'b':1, 'c':2}
```

Dictionaries and Tuples

the dictionary view object, dict.items() is a sequence of tuples.

In [86]:
```
d = {'a':0, 'b':1, 'c':2}
```

In [87]:
```
d.items()
```

Out[87]:
```
dict_items([('a', 0), ('b', 1), ('c', 2)])
```

Dictionaries and Tuples

the dictionary view object, dict.items() is a sequence of tuples.

In [86]:
```
d = {'a':0, 'b':1, 'c':2}
```

In [87]:
```
d.items()
```

Out[87]:
```
dict_items([('a', 0), ('b', 1), ('c', 2)])
```

In [88]:
```
for key, value in d.items():
    print(key, value)
```

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

In [89]:

```
d = dict(enumerate("efg"))
```

In [89]:
```
d = dict(enumerate("efg"))
```

In [90]:
```
d
```

Out[90]:
```
{0: 'e', 1: 'f', 2: 'g'}
```

In [89]:
```
d = dict(enumerate("efg"))
```

In [90]:
```
d
```

Out[90]:
```
{0: 'e', 1: 'f', 2: 'g'}
```

Swap the keys and elements in a dictionary

In [89]:
```
d = dict(enumerate("efg"))
```

In [90]:
```
d
```

Out[90]:
```
{0: 'e', 1: 'f', 2: 'g'}
```

Swap the keys and elements in a dictionary

In [91]:
```
swapped = {}
for key, value in d.items():
    swapped[value] = key
```

In [89]:
```
d = dict(enumerate("efg"))
```

In [90]:
```
d
```

Out[90]:
```
{0: 'e', 1: 'f', 2: 'g'}
```

Swap the keys and elements in a dictionary

In [91]:
```
swapped = {}
for key, value in d.items():
    swapped[value] = key
```

In [92]:
```
swapped
```

Out[92]:
```
{'e': 0, 'f': 1, 'g': 2}
```

In [93]:
```
dict(zip("efg", range(3)))
```

Out[93]:
```
{'e': 0, 'f': 1, 'g': 2}
```

We can create dictionaries out of sequences of tuples and with zip objects

We can create dictionaries out of sequences of tuples and with zip objects

In [94]:
```
l = [('z', 25), ('y', 24), ('x', 23)]
dict(l)
```

Out[94]:
```
{'z': 25, 'y': 24, 'x': 23}
```

We can create dictionaries out of sequences of tuples and with zip objects

In [94]:
```
l = [('z', 25), ('y', 24), ('x', 23)]
dict(l)
```

Out[94]:
```
{'z': 25, 'y': 24, 'x': 23}
```

In [95]:
```
z = zip('xyz', [23, 24, 25])
dict(z)
```

Out[95]:
```
{'x': 23, 'y': 24, 'z': 25}
```

Tuples as dictionary keys

Because tuples are immutable, they can be used as keys in a dictionary

For example, there might be a 2D function that is very expensive to compute for coordinates. You can create a dictionary that will store all of the values that have been calculated for each 2D pair.

Let's say you have a function: $f(x, y) = x^2 + 2y$

Tuples as dictionary keys

Because tuples are immutable, they can be used as keys in a dictionary

For example, there might be a 2D function that is very expensive to compute for coordinates. You can create a dictionary that will store all of the values that have been calculated for each 2D pair.

Let's say you have a function: $f(x, y) = x^2 + 2y$

In [96]:
```
# this dictionary contains values that are known solutions
known = {(0, 0): 0}
```

Tuples as dictionary keys

Because tuples are immutable, they can be used as keys in a dictionary

For example, there might be a 2D function that is very expensive to compute for coordinates. You can create a dictionary that will store all of the values that have been calculated for each 2D pair.

Let's say you have a function: $f(x, y) = x^2 + 2y$

In [96]:
```
# this dictionary contains values that are known solutions
known = {(0, 0): 0}
```

In [97]:
```
def f(x, y):
    t = (x,y)
    if t in known:
        print("value already exists dictionary")
        return known[t]
    print("value must be calculated")
    res = x ** 2 + 2 * y
    known[t] = res
    return res
```

In [98]:
```
f(0, 0)
```

```
value already exists dictionary
```

Out[98]:
```
0
```

In [98]:
```
f(0, 0)
```

```
value already exists dictionary
```

Out[98]:
```
0
```

In [99]:
```
f(1, 2)
```

```
value must be calculated
```

Out[99]:
```
5
```

In [98]:
```
f(0, 0)
```

```
value already exists dictionary
```

Out[98]:
```
0
```

In [99]:
```
f(1, 2)
```

```
value must be calculated
```

Out[99]:
```
5
```

In [100]:
```
f(1, 2)
```

```
value already exists dictionary
```

Out[100]:
```
5
```

In [98]:
```
f(0, 0)
```

```
value already exists dictionary
```

Out[98]:
```
0
```

In [99]:
```
f(1, 2)
```

```
value must be calculated
```

Out[99]:
```
5
```

In [100]:
```
f(1, 2)
```

```
value already exists dictionary
```

Out[100]:
```
5
```

In [101]:
```
known
```

Out[101]:
```
{(0, 0): 0, (1, 2): 5}
```