

Week 2 - Lists, Tuples, Sets, and Dictionaries

Collections

So far we have worked mostly with individual objects - individual numbers, individual strings, and so on. But it is very common, when programming, to work with whole **collections** of objects - collections of numbers, collections of strings, even mixed collections of objects of different types.

As programming languages have developed, collections have proven to be a cornerstone of working with data, and Python has some of the most powerful techniques for manipulating collections among modern programming languages.

Python provides four built-in types of collection, each useful in their own way:

- **Lists**
- **Tuples**
- **Sets**
- **Dictionaries**

Lists

A list is an ordered collection of objects (possibly empty). The objects can be of any type, and they can be repeated.

Tuples

A tuple is like a list except that it is *immutable*, which means that objects cannot be added to a tuple, they cannot be removed from a tuple, and they cannot be reordered within a tuple.

If it's important that the collection does not change then you should use a tuple, even if the careful use of a list would achieve the same thing. By using a tuple you guarantee that it won't change, and you also signal your intentions more clearly to anyone who reads your code.

Sets

A set is an *unordered* collection of objects (possibly empty), each of which is *unique* (cannot be repeated).

If it's important that the collection does not contain the same object twice then you should use a set, even if the careful use of a list would achieve the same thing. By using a set you guarantee that there won't be duplicates, and you also signal your intentions more clearly to anyone who reads your code.

Dictionaries

A dictionary is an *ordered* collection of *key-value pairs*. In addition, the keys must be unique. You can think of a dictionary as a *mapping* from a set of keys to some values. You'll learn more about what this means as you work your way through this week's slides.



In earlier versions of Python dictionaries were not ordered. They have been ordered since version 3.7.

Creating a collection

You can create a collection in a variety of ways.

List literals

A list literal is a sequence of literals, each of which refers to an object, separated by commas, and surrounded by square brackets. For example:

```
x = [2, 4, 6, 8]
x = ['cat', 'mouse', 'cat', 'mouse']
x = [1, 3.14, 'a', True]
x = []
```

Note the last literal above - it is the literal for the **empty list**. Also note that the same object can be included more than once in a list.

The expressions that you use within square brackets need not be literals - they could be variables or other complex expressions. For example:

```
y = 45
x = [2*23, 4-1, abs(-6), y]
print(x)
```

But, strictly speaking, if they are not literals then the whole thing does not count as a list literal - it's only a list literal if all of the expressions in the square brackets are themselves literals.

Tuple literals

A tuple literal is like a list literal, except you use round brackets rather than square brackets:

```
x = (2, 4, 6, 8)
x = ('cat', 'mouse', 'dog')
x = (1, 3.14, 'a', True)
x = ()
```

Note the last literal again - it is the literal for the **empty tuple**.

In some contexts you can drop the round brackets around a tuple. We have already seen an example of this - variable unpacking. For example, line 1 below is just shorthand for line 2:

```
x, y = 1, 2
(x, y) = (1, 2)
```

You should be careful when doing this, because it sometimes causes errors, and it can make your

code less readable.

Set literals

A set literal is also like a list literal, except you use curly brackets rather than square brackets:

```
x = {2, 4, 6, 8}
x = {'cat', 'mouse', 'dog'}
x = {1, 3.14, 'a', True}
```

What about the empty set? You might expect that you could use `{}`, but unfortunately that refers to the empty *dictionary* (see below). If you want an empty set you have to use `set()` (more on this below).

The items in a set are *unique*, so what happens if you try to include an object more than once? Try it:

```
x = {'cat', 'mouse', 'dog', 'cat', 'cat'}
print(x)
```

As you can see, Python silently ignores all but one occurrence of the object.

Dictionary literals

A dictionary literal uses curly brackets, like set literals, but with key-value pairs separated by commas, and each key and value separated by a colon.

```
x = {1: 'cat', 2: 'dog', 3: 'mouse'}
x = {'A': 0, 'B': 1, 'E': 2, 'M': 3}
x = {}
```

Note the last literal again - it is the literal for the **empty dictionary**.

Although you'll mostly use integers, floats, and strings for the keys of a dictionary, they can also be booleans or tuples:

```
x = {True: 1, False: 0}
print(x)
x = {(0, 0): 'bottom left', (1, 1): 'top right'}
print(x)
```

They cannot be lists or sets or dictionaries. Try it:

```
try: x = {[1, 2]: 'First list', [3, 4]: 'Second list'}
except Exception as e: print(e)
try: x = {{1, 2}: 'First set', {3, 4}: 'Second set'}
except Exception as e: print(e)
try: x = {{1: 2}: 'First dictionary', {3: 4}: 'Second dictionary'}
except Exception as e: print(e)
```

The problem with lists, sets and dictionaries is that they are mutable, which means that Python cannot **hash** them (which means, roughly, giving them a unique unchanging value).

The keys do not all have to be the same type, and neither do the values:

```
x = {1: 'one', 'two': 2, 3.176: False}
print(x)
```

However, mixing types can lead to confusion, and it is good programming practice to keep the types the same.

In a dictionary, the values can be duplicated but the keys must be unique. So what happens if you use the same key twice? Try it:

```
x = {'A': 0, 'B': 1, 'A': 2, 'B': 3}
print(x)
```

As you can see, if there are multiple items with the same key then Python silently ignores all but the last one.

Collections of collections

The items in a collection can be objects of any type. In particular, they can be collections. So it's possible to create collections of collections. For example:

```
# A list of lists
x = [[1, 2, 3], [3, 4, 5], [6, 7, 8]]
print(x)

# A set of tuples
x = {(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')}
print(x)
```

Using other collections

You can also use the `list()`, `tuple()`, `set()`, and `dict()` functions to create a collection of one type from a collection of another type.

Using `list()`

You can create a list from a tuple, set, or dictionary:

```
x = list((1, 2, 3)) # From a tuple
print(x)
x = list({1, 2, 3}) # From a set
print(x)
x = list({1: 'a', 2: 'b', 3: 'c'}) # From a dictionary
print(x)
```

Notice that when you create a list from a dictionary you get a list of the *keys*.

Using `tuple()`

You can create a tuple from a list, set, or dictionary:

```
x = tuple([1, 2, 3]) # From a list
print(x)
x = tuple({1, 2, 3}) # From a set
print(x)
x = tuple({1: 'a', 2: 'b', 3: 'c'}) # From a dictionary
print(x)
```

Using `set()`

You can create a set from a list, tuple, or dictionary:

```
x = set([1, 2, 3]) # From a list
print(x)
x = set((1, 2, 3)) # From a tuple
print(x)
x = set({1: 'a', 2: 'b', 3: 'c'}) # From a dictionary
print(x)
```

Creating a set from a list can be a good way to remove duplicate items from the list:

```
x = [1, 2, 2, 1, 3, 3, 2, 3, 1]
print(x)
x = set(x)
print(x)
```

If you want the result as a list then you can simply apply the `list` function after applying the `set` function:

```
x = [1, 2, 2, 1, 3, 3, 2, 3, 1]
x = list(set(x))
print(x)
```

Using `dict()`

Since a dictionary is a collection of key-value *pairs*, you can't create a dictionary from a list, tuple, or set unless the items are pairs. But when they are you have quite a bit of flexibility. Any of the following will work:

```
x = dict([[1, 'a'], [2, 'b']]) # From a list of lists
print(x)
x = dict([(1, 'a'), (2, 'b')]) # From a list of tuples
print(x)
x = dict((1, 'a'), (2, 'b')) # From a tuple of lists
```

```
print(x)
x = dict(((1, 'a'), (2, 'b')))) # From a tuple of tuples
print(x)
x = dict({(1, 'a'), (2, 'b')}) # From a set of tuples
print(x)
```

Be careful when retrieving the items of a set, because you can't always tell in what order Python will give them to you. For example, the following is risky, because you can't tell in advance what the keys will be and what the values will be:

```
x = dict([{1, 'a'}, {2, 'b'}])
print(x)
```

You should avoid using sets in this way.

Inspecting a collection

There are a variety of ways in which you can inspect a collection.

Number of items

You can get the number of items in a collection using Python's `len()` function:

```
print(len([1, 2, 3])) # List
print(len((1, 2, 3))) # Tuple
print(len({1, 2, 3})) # Set
print(len({'a': 1, 'b': 2, 'c': 3})) # Dictionary
print(len([])) # Empty list
```

Notice that if you have a collection of collections, `len()` only counts the number of items at the top level:

```
numbers = [[1, 2, 3], [3, 4, 5], [5, 6, 7]]
print(len(numbers))
```

Existence of an item

You can check whether a collection contains an item using the `in` keyword. Note that in the case of dictionaries it is the keys that are checked.

```
print(1 in [1, 2, 3]) # True
print('a' in [1, 2, 3]) # False

print(1 in (1, 2, 3)) # True
print('a' in (1, 2, 3)) # False

print(1 in {1, 2, 3}) # True
print('a' in {1, 2, 3}) # False

print(1 in {1: 'a', 2: 'b', 3: 'c'}) # True - keys are checked
print('a' in {1: 'a', 2: 'b', 3: 'c'}) # False - keys are checked
```

You can check whether an item is *not* in a collection either by using `in` and checking whether `False` is returned, or by using `not in` and checking whether `True` is returned:

```
print('a' in [1, 2, 3]) # False
print('a' not in [1, 2, 3]) # True
```

You can use the `index()` method to find the index of the first occurrence of a value in a list or tuple. If the value is not found then Python raises an error.


```
x = [1, 3, 7, 8, 7, 5, 4, 6, 8, 5, 7]
print(x.index(7)) # Only the first occurrence is found
```

Frequency of an item

If you want to know how many times an item occurs in a list or tuple you can use the `count()` method (sets and dictionaries do not have this method):

```
x = [1, 1, 2, 2, 2]
print(x.count(1))
x = (1, 1, 2, 2, 2)
print(x.count(1))
print(x.count('a'))
```

You could also use this to check whether a list or tuple contains a certain item (if not, the count will be zero).

Minimum item

You can find the minimum item in a collection by using the `min()` function. In the case of dictionaries it compares their *keys*, rather than their values.

```
print(min([1, 2, 3])) # List
print(min((1, 2, 3))) # Tuple
print(min({1, 2, 3})) # Set
print(min({'a': 10, 'b': 5, 'c': 1})) # Dictionary - compares the keys
```

Maximum item

You can find the maximum item in a collection by using the `max()` function. Again, in the case of dictionaries it compares their keys.

```
print(max([1, 2, 3])) # List
print(max((1, 2, 3))) # Tuple
print(max({1, 2, 3})) # Set
print(max({'a': 10, 'b': 5, 'c': 1})) # Dictionary - compares the keys
```

Sum of the items

You can find the sum of the items in a collection by using the `sum()` function. In the case of dictionaries it sums their keys (so there will be an error if the keys cannot be summed, such as when they are letters).

```
print(sum([1, 2, 3])) # List
print(sum((1, 2, 3))) # Tuple
print(sum({1, 2, 3})) # Set
print(sum({1: 'a', 2: 'b', 3: 'c'})) # Dictionary - sums the keys
```

Checking if all items are true

You can check whether *all* the items in a collection evaluate to `True` using the `all` function. As you probably expect by now, in the case of dictionaries it checks the keys, not the values.

```
print(all([True, True]))
print(all([True, False]))
print(all(('a', 'b')))
print(all('', 'b'))
print(all({1, 2}))
print(all({0, 1}))
print(all({1: 'a', 2: 'b', 3: 'c'}))
print(all({0: 'a', 1: 'b', 1: 'c'}))
```

Checking if any items are true

You can check whether *any* of the items in a collection evaluate to `True` using the `any` function. Again, in the case of dictionaries it checks the keys, not the values.

```
print(any([True, False]))
print(any([False, False]))
print(any('', 'b'))
print(any('', 0))
print(any({1, 0}))
print(any({0, 0}))
print(any({0: 'a', 1: 'b', 2: 'c'}))
print(any({0: 'a', '': 'b', False: 'c'}))
```

Selecting elements

Lists and tuples

The items of a list or tuple are ordered, so each one has a position in the list. An item's position is also called its **index**. The indexes are integers, **starting from 0**. So, the first item has index 0, the second item has index 1, and so on.

You can select items in a list or tuples by using the **indexing operator** `[]` :

```
x = ['a', 'b', 'c', 'd', 'e']
print(x[0])
print(x[3])
```

Indexes can also be negative. The last item has index -1, the second last item has index -2, and so on.

```
x = ['a', 'b', 'c', 'd', 'e']
print(x[-1])
print(x[-2])
```

You can also use the indexing operator to get a **slice** of a list or tuple. The syntax for slicing is `[start:end]`, where `start` is the start index and `end` is the end index. The cuts are made just *before* the items at the start and end indexes, so the slice includes the start item but not the end item.

```
x = ['a', 'b', 'c', 'd', 'e']
print(x[0:3]) # Prints items with indexes 0, 1, 2
```

Notice that `x[0:3]` returns items with index 0 to 2, rather than 0 to 3 as you might expect - this can be very confusing!

You can also use negative indices when slicing:

```
x = ['a', 'b', 'c', 'd', 'e']
print(x[1:-1])
```

If you don't specify a start index then it is assumed to be zero. If you don't specify an end index then it is assumed to be the length of the list (an invalid index, but as it is excluded - this is fine and is the way to include the last element of the list). If you don't specify either then the whole list is returned.

```
x = ['a', 'b', 'c', 'd', 'e']
print(x[:3])
print(x[1:])
print(x[:])
```

Python allows you to add a third parameter to the slice to control the **step**. This parameter allows

you to easily extract every *nth* element from the list. The syntax for slicing with a step is `[start:end:step]`, where `step` is an integer.

```
x = ['a', 'b', 'c', 'd', 'e']
print(x[::2]) # Every second element
print(x[::3]) # Every third element
print(x[::-1]) # Every element in reverse
print(x[::-2]) # Every second element in reverse
```

Getting a random item

It can be useful to get a random item of a collection. For lists and tuples you can use the `choice()` function of the `random` module:

```
import random
print(random.choice([1, 2, 3, 4, 5]))
print(random.choice((1, 2, 3, 4, 5)))
```

Sets

The items in a set are *unordered*, which means that they have no index. If you try to refer to an item in a set by using the indexing operator (square brackets notation) you will get an error:

```
x = {'cat', 'mouse', 'dog'}
print(x[0]) # Error
```

Dictionaries

You can access the elements of a dictionary also using the indexing operator, but in this case the indices are the *keys* of the dictionary, which are not necessarily integers.

```
scores = {
    'Alice': 0,
    'Bob': 1,
    'Eve': 2,
    'Mallory': 3,
}
print(scores['Alice'])
```

If there is no element with the given index then an error will occur:

```
scores = {
    'Alice': 0,
    'Bob': 1,
    'Eve': 2,
    'Mallory': 3,
}
print(scores['Steve'])
```

To allow for this you can use the `get()` method instead, which allows you to specify a default value in case there is no matching index:

```
scores = {  
    'Alice': 0,  
    'Bob': 1,  
    'Eve': 2,  
    'Mallory': 3,  
}  
print(scores.get('Steve', 'There is no such index'))
```

Note that lists and tuples do not have a `get()` method.

Adding elements

Adding to a list

You can add an item to the end of a list by using the `append()` method:

```
x = ['a', 'b', 'c', 'd', 'e']
x.append('f')
print(x)
```

Notice that the `append()` method modifies the list **in-place** - you do not have to assign the result back to the variable. This is different from the string methods we saw last week, which do not modify strings in place, but return new values:

```
x = "Hello"
x.upper() # x not changed
print(x)
x = x.upper() # x changed
print(x)
```

You can insert an item at a specified index by using the `insert()` method:

```
x = ['a', 'b', 'c', 'd', 'e']
x.insert(2, 'x') # Insert at index 2
print(x)
```

You can achieve the same thing by replacing the empty slice from 2 to 2:

```
x = ['a', 'b', 'c', 'd', 'e']
x[2:2] = 'x' # Insert at index 2
print(x)
```

You can extend a list with the items from another list by using the `extend()` method:

```
x = ['a', 'b', 'c', 'd', 'e']
y = ['x', 'y', 'z']
x.extend(y)
print(x)
```

Notice that the `extend()` method also modifies the list in-place - you do not have to assign the result back to the variable.

Also notice that extending by a list is different from appending a list. Compare the result of extending (above) with the result of appending (below):

```
x = ['a', 'b', 'c', 'd', 'e']
```

```
y = ['x', 'y', 'z']
x.append(y)
print(x)
```

You can also extend a list with the items from another list using the `+` operator:

```
x = ['a', 'b', 'c', 'd', 'e']
y = ['x', 'y', 'z']
x = x + y
print(x)
```

Notice that the `+` operator does not modify the list in-place - you have to assign the result back to the variable.

Adding to a tuple

Because tuples are immutable you cannot add items to them.

Adding to a set

You can add a single item to a set by using the set's `add()` method. If the item is already in the set, Python quietly ignores the request.

```
letters = {'a', 'b'}
letters.add('c')
print(letters)
letters.add('c') # No error, just not added
print(letters)
```

You can add multiple items to a set by using the set's `update()` method:

```
vowels = set()
vowels.update('a', 'e')
print(vowels)
```

Adding to a dictionary

You can add an item to a dictionary by specifying a value for a new key. If the key already exists, the value for that key will be updated.

```
x = {1: 'a', 2: 'b'}
x[3] = 'c' # Item added
print(x)
x[3] = 'd' # Item updated
print(x)
```

Removing elements

Removing from a list

You can remove an item at a specific index using a `del` statement:

```
x = ['a', 'b', 'c', 'd', 'e']
del x[1]
print(x)
```

You can do the same with a slice of the list:

```
x = ['a', 'b', 'c', 'd', 'e']
del x[1:3]
print(x)
```

You can also remove a slice by setting it to the empty list:

```
x = ['a', 'b', 'c', 'd', 'e']
x[1:3] = [] # Items removed
print(x)
```

This doesn't work for individual items:

```
x = ['a', 'b', 'c', 'd', 'e']
x[0] = [] # Item not removed - replaced by the empty list
print(x)
```

You can also remove an item at a specific index using the `pop()` method. If `pop()` isn't given an index then the last item will be removed. The removed element is returned.

```
x = ['a', 'b', 'c', 'd', 'e']
print(x.pop(2)) # Removes and returns item at index 2
print(x)
print(x.pop()) # Removes and returns the last item
print(x)
```

You can remove the first item with a given value by using the `remove()` method:

```
x = ['a', 'b', 'c', 'd', 'e', 'c']
x.remove('c') # Only first occurrence is removed
print(x)
```

You can remove all items from a list by using the `clear()` method:

```
x = ['a', 'b', 'c', 'd', 'e']
```



```
x.clear() # All items removed
print(x)
```

Removing from a tuple

Because tuples are immutable you cannot remove items from them.

Removing from a set

Because a set is not ordered you cannot remove elements by index, but you can remove them by value, using the set's `remove()` or `discard()` methods. If the item is not in the set then using `remove()` will cause an error, but using `discard()` will not:

```
vowels = {'a', 'e', 'i', 'o', 'u'}
vowels.remove('a') # Removes 'a'
print(vowels)
vowels.discard('f') # No error
print(vowels)
vowels.remove('f') # Error
```

You can remove all items from a set by using the set's `clear()` method:

```
vowels = {'a', 'e', 'i', 'o', 'u'}
vowels.clear() # Remove all items
print(vowels)
```

Removing from a dictionary

You can remove an element of a dictionary by key using `del` or `pop()`:

```
scores = {'A': 0, 'B': 1, 'E': 2, 'M': 3}
del scores['B'] # Remove item whose key is 'B'
print(scores)
scores.pop('M') # Remove item whose key is 'M', and return its value
print(scores)
```

You can remove all elements using `clear()`:

```
scores = {'A': 0, 'B': 1, 'E': 2, 'M': 3}
scores.clear() # Remove all items
print(scores)
```

Modifying elements

Modifying list elements

You can change the value of an item in a list by referring to it and then assigning it a new value:

```
x = ['a', 'b', 'c', 'd', 'e']
x[0] = 'z' # Assign a new value
print(x)
```

You can also assign to a slice with a sequence:

```
x = ['a', 'b', 'c', 'd', 'e']
x[0:1] = ['x', 'y'] # Assign new values
print(x)
```

Modifying tuple elements

Since tuples are immutable, you cannot change which objects are in the tuple:

```
x = (0, 2, 3)
x[0] = 1 # Error - cannot change which objects are in the tuple
```

But if one of those objects is itself mutable, then you can change the nature of that object. Suppose the first element of a tuple is a list, for example. You can change the elements of this list, even though the list is part of a tuple. That's because you're not changing *which* objects are in the tuple - you're just changing the nature of one of those objects. It's not a good idea to do this, though - it goes against the spirit of using tuples to signal that your data should not change.

Modifying set elements

You can't change an element of a set, but you remove it and then add a different element:

```
x = {1, 2, 3}
x.remove(3)
x.add(4)
print(x)
```

Modifying dictionary elements

Updating an item in a dictionary is similar to updating an item in a list.

```
scores = {
    'Alice': 0,
    'Bob': 1,
```

```
        'Eve': 2,  
        'Mallory': 3,  
    }  
    scores['Bob'] = 900  
    scores['Alice'] += 1  
    print(scores)
```

Sorting elements

Sorting a list

You can sort the elements of a list by using the `sort()` method, which orders the items by comparing their values using the `<` operator. Note that this method sorts the list **in-place** - it does not return a new list.

```
x = [1, 5, 4, 2, 3]
x.sort()
print(x)

x = ['c', 'a', 'e', 'b', 'd']
x.sort()
print(x)
```

You can sort the elements in descending order by passing the keyword argument `reverse = True`:

```
x = [1, 5, 4, 2, 3]
x.sort(reverse = True)
print(x)
```

You could also achieve this by sorting them in ascending order and then reversing the list, using the `reverse()` method:

```
x = [1, 5, 4, 2, 3]
x.sort()
x.reverse()
print(x)
```

Note that reversing a list is not the same thing as sorting it in descending order:

```
x = [1, 5, 4, 2, 3]
x.sort(reverse = True)
print(x)
x = [1, 5, 4, 2, 3]
x.reverse()
print(x)
```

Also note that if you have a list of lists the `reverse()` method only reverses the topmost level of lists:

```
numbers = [[1, 2, 3], [3, 4, 5], [5, 6, 7]]
numbers.reverse()
print(numbers)
```

Sorting uppercase and lowercase

Somewhat surprisingly, Python sorts all uppercase characters before all lowercase characters. So 'B' comes before 'a'. Try it:

```
x = ['a', 'A', 'b', 'B', 'c', 'C']
x.sort()
print(x)
```

The reason for this is that Python sorts characters according to their ASCII numeric value, and the ASCII values of uppercase characters are lower than the ASCII values of lowercase characters. You can see this, by using the `ord()` function:

```
print(ord('a'))
print(ord('A'))
print(ord('b'))
print(ord('B'))
print(ord('c'))
print(ord('C'))
```

Sorting with a function

Suppose you have a list of words and you want to sort them by length. If you use the bare `sort()` method then you will get the wrong result - it will sort them alphabetically:

```
x = ['dog', 'chicken', 'mouse', 'horse', 'goat', 'donkey']
x.sort()
print(x)
```

In this case you need to specify a `key`, which is a function that returns, for each item in the list, the value that you'd like to sort the item by. In this case we'd like to sort items by length, so we can use the `len()` function:

```
x = ['dog', 'chicken', 'mouse', 'horse', 'goat', 'donkey']
x.sort(key = len)
print(x)
```

Later you will learn how to define your own functions. This will allow you to do even more sophisticated sorting.

What about tuples, sets and dictionaries?

Since tuples are immutable they cannot be sorted, so tuples do not have a `sort()` method.

Since the elements in a set are not ordered it doesn't make sense to sort them (you cannot be guaranteed of getting them back in any particular order), so sets do not have a `sort()` method.

Although the elements of a dictionary are ordered, they are always in **insertion order** - the order in

which they were inserted into the dictionary. So it doesn't make sense to sort a dictionary, and dictionaries do not have a `sort()` method.

You can, however, create a sorted *list* from these things, using Python's `sorted()` function.

The sorted function

The `sorted()` function takes a collection and returns a list of its items sorted in ascending order. It leaves the original collection unchanged (so it can be used on tuples). Note that in the case of a dictionary it returns the keys of the dictionary, sorted.

```
x = (1, 5, 4, 2, 3) # Tuple
print(sorted(x)) # Returns a new list, sorted

x = {1, 5, 4, 2, 3} # Set
print(sorted(x)) # Returns a new list, sorted

x = {1:'f', 5:'a', 4:'b', 2:'d', 3:'c'} # Dictionary
print(sorted(x)) # Returns a new list, from the keys, sorted
```

If you want the resulting list in descending order, pass the argument `reverse = True`:

```
x = (1, 5, 4, 2, 3)
print(sorted(x, reverse = True))

x = {1, 5, 4, 2, 3} # Set
print(sorted(x, reverse = True))

x = {1:'f', 5:'a', 4:'b', 2:'d', 3:'c'}
print(sorted(x, reverse = True))
```

Joining elements

You can join the elements of a collection into a string, as long as they are themselves strings. The method by which you do this is somewhat counterintuitive:

```
x = ['a', 'b', 'c', 'd', 'e']
s = ','.join(x)
print(s)
```

To join the elements of the list into a string using commas to separate the items, you call the `join()` method on the comma string, and supply the list as an argument.

You can use whatever separator you want:

```
x = ['a', 'b', 'c', 'd', 'e']
print(', '.join(x))
print(' '.join(x))
print('--'.join(x))
print(' then '.join(x))
```

This technique works for tuples, sets and dictionaries too (in the case of dictionaries it is the keys that get joined, and they must be strings):

```
x = ('a', 'b', 'c', 'd', 'e') # Tuple
print('-'.join(x))

x = {'a', 'b', 'c', 'd', 'e'} # Set
print('-'.join(x))

x = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5} # Dictionary
print('-'.join(x))
```

Special string operations

Strings as tuples

In Python we can think of a string as **a tuple of characters**.

This means that you can access the characters in a string in the same ways that you can access the items in a tuple, using the `[]` indexing operator. You can select individual characters this way, and you can select slices of characters, which are substrings:

```
x = 'Hello'
print(x[1]) # Individual character
print(x[1:3]) # Slice
print(x[2:-1]) # Slice
```

But you cannot reassign individual characters:

```
x = 'Hello'
x[1] = 'J'
print(x) # Error
```

Splitting strings

One of the most useful methods available for string objects is the `split()` method. This breaks a string into a **list of substrings** that are separated by a delimiter.

```
y = 'The cat sat on the mat'
print(y.split(' ')) # Split using ' ' as the delimiter

z = '12:30:45'
print(z.split(':')) # Split using ':' as the delimiter
```

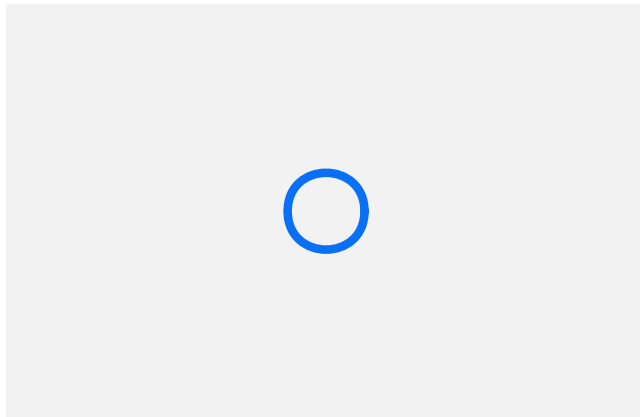
Note that the delimiter is not included in the substrings.

Special set operations

There are special operations that you can perform on sets.

Union

The **union** of two sets is the set of items that belong to *either or both* sets:

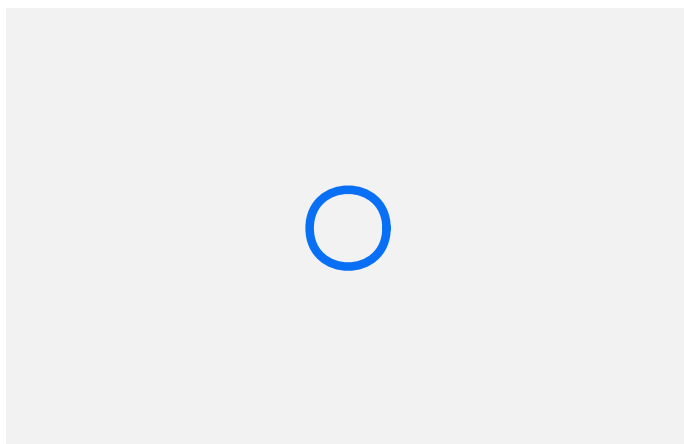


You can get the union of two sets using either the `|` operator or the `union()` method:

```
evens = {2, 4, 6, 8}
primes = {2, 3, 5, 7}
print(evens | primes)
print(evens.union(primes))
print(primes.union(evens))
```

Intersection

The **intersection** of two sets is the set of items that belong to *both* sets:



You can get the intersection of two sets using either the `&` operator or the `intersection()` method:

```
evens = {2, 4, 6, 8}
primes = {2, 3, 5, 7}
```

```
print(evens & primes)
print(evens.intersection(primes))
print(primes.intersection(evens))
```

Difference

The **difference** between set A and set B is the set of items that belong to A but not B:



You can get the difference between two sets using either the `-` operator or the `difference()` method:

```
evens = {2, 4, 6, 8}
primes = {2, 3, 5, 7}
print(evens - primes)
print(evens.difference(primes))
print(primes - evens)
print(primes.difference(evens))
```

Symmetric difference

The **symmetric difference** between set A and set B is the set of items that belong to A but not B, or to B but not A.

You can get the symmetric difference between two sets using the `symmetric_difference()` method:

```
evens = {2, 4, 6, 8}
primes = {2, 3, 5, 7}
print(evens.symmetric_difference(primes))
print(primes.symmetric_difference(evens))
```

Comparing sets

Two sets are **disjoint** when they have no elements in common.

You can check whether two sets are disjoint using the `isdisjoint()` method:

```
evens = {2, 4, 6, 8}
```

```
primes = {2, 3, 5, 7}
print(evens.isdisjoint(primes))
print(primes.isdisjoint(evens))
```

A set A is a **subset** of a set B when every element of A is also an element of B. We also say, in this case, that B is a **superset** of A.

You can check whether one set is a subset of another using either the `<=` operator or the `issubset()` method:

```
A = {2, 3}
B = {2, 3, 6, 7}
C = {4, 5, 6, 7}
print(A.issubset(B))
print(A <= B)
print(A.issubset(C))
print(A <= C)
```

Similarly, you can check whether one set is a superset of another using either the `>=` operator or the `issuperset()` method:

```
A = {2, 3}
B = {2, 3, 6, 7}
C = {4, 5, 6, 7}
print(B.issuperset(A))
print(B >= A)
print(C.issuperset(A))
print(C >= A)
```

Looping through a collection

For loops

It is very common to loop through the items in a collection one-by-one. A good way to do so is to use a `for` loop, which is custom-made for this kind of thing:

```
vowels = ['a', 'e', 'i', 'o', 'u']
for v in vowels:
    print(v)
```

You can use this technique not just with lists but with tuples, sets and dictionaries too:

```
for x in (1, 2, 3): # Tuple
    print(x)
for x in {1, 2, 3}: # Set
    print(x)
for x in {1:'a', 2:'b', 3:'c'}: # Dictionary
    print(x)
```

Keep in mind that because sets are not ordered there is no guarantee in what order their items will be looped through. Try running the following piece of code a few times:

```
for x in {'a', 'e', 'i', 'o', 'u'}:
    print(x)
```

Also notice that when you loop through a dictionary it is the *keys* that get looped through:

```
for x in {1: 'a', 2: 'e', 3: 'i', 4: 'o', 5: 'u'}:
    print(x)
```

Variations

Sometimes when you loop through a list you will want to use the index of an item as well as its value. You can do this by using the `enumerate()` function, which returns a collection of key-value pairs for you to loop through:

```
vowels = ['a', 'e', 'i', 'o', 'u']
for index, vowel in enumerate(vowels):
    print('The vowel at index', index, 'is', vowel)
```

You can do a similar thing with dictionaries by using the `items()` method of a dictionary:

```
scores = {'Alice': 0, 'Bob': 1, 'Eve': 2, 'Mallory': 3}
for key, value in scores.items():
```

```
print(f'{key} scored {value}')
```

If you want to loop through just the values of a dictionary you can use the `values()` method:

```
scores = {'Alice': 0, 'Bob': 1, 'Eve': 2, 'Mallory': 3}
for x in scores.values():
    print(x)
```

Break and continue

In Week 1 you learned about using `break` and `continue` in a while loop. They can also be used in `for` loops.

```
vowels = ['a', 'e', 'i', 'o', 'u']
```

```
print('Everything before o:')
for v in vowels:
    if v == 'o':
        break
    print(v)
```

```
print('Everything except o:')
for v in vowels:
    if v == 'o':
        continue
    print(v)
```

Nested loops

If you are looping through a collection whose items are themselves collections then you might want to use **nested** loops.

Suppose that you have a list of lists of numbers, and you want to add up all the numbers. You could do this using nested loops:

```
lists = [[1, 2, 3], [3, 4, 5], [5, 6, 7]]
total = 0
for lst in lists:
    for num in lst:
        total += num
print(total)
```

Comprehensions

List comprehensions

Suppose you have a list of words and you'd like a list of their lengths. You could get it by using a `for` loop:

```
words = ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
lengths = []
for word in words:
    lengths.append(len(word))
print(lengths)
```

Python provides a more elegant way to do this - a **list comprehension**:

```
words = ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
lengths = [len(word) for word in words]
print(lengths)
```

The expression `[len(word) for word in words]` is the list comprehension. Note that it is an expression, rather than a block of statements.

You can add a condition to the comprehension. Suppose you only want to include words that are more than three letters long. Then you could use the following:

```
words = ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
lengths = [len(word) for word in words if len(word) > 3]
print(lengths)
```

This is a very useful way of filtering out elements from a list. Here's another example:

```
nums = [2, 12, 4, 2, 9, 10, 11, 1, 15, 23]
nums = [x for x in nums if x >= 10]
print(nums)
```

In a previous example we used nested `for` loops to sum the numbers in a list of lists of numbers. We can do it using list comprehensions:

```
lists = [[1, 2, 3], [3, 4, 5], [5, 6, 7]]
total = sum(sum(x) for x in lists)
print(total)
```

Set comprehensions

If you'd like a *set* of lengths rather than a list of lengths then you could use a **set comprehension**,

which is just the same but with curly brackets:

```
words = ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
unique_lengths = {len(word) for word in words}
print(unique_lengths)
```

Note that duplicate values are automatically removed from the set.

Dictionary comprehensions

You can also create a dictionary using a **dictionary comprehension**. Here's a slightly different example:

```
words = ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
words_with_lengths = {word: len(word) for word in words}
print(words_with_lengths)
```

What about a tuple comprehension?

Alas, there is no tuple comprehension. You can use the same sort of construction, but it will give you a generator, not a tuple (more about generators next week).

```
words = ['the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
lengths = (len(word) for word in words)
print(lengths)
```

The range function

A very common thing to do when programming is to loop through a sequence of numbers. Python's `range()` function is a very useful way of creating the numbers to loop through.

```
for i in range(10):  
    print(i)
```

Note that `range(n)` returns `n` integers, from `0` to `n-1`. The number `n` is not included.

Here's how you might use it:

```
vowels = ['a', 'e', 'i', 'o', 'u']  
for i in range(len(vowels)):  
    print('The vowel at index', i, 'is', vowels[i])
```

Although, you might find it more convenient in this case to use the `enumerate()` function:

```
vowels = ['a', 'e', 'i', 'o', 'u']  
for i, value in enumerate(vowels):  
    print('The vowel at index', i, 'is', value)
```

What range returns

It might seem like the range function returns these numbers as a list, the list `[0, 1, ..., 10]`. But actually it doesn't. It returns a special type of object called a **range**. You see this by checking its type:

```
x = range(10)  
print(type(x))
```

A range object is a method for generating each number as required, but not until it is required.

If you'd like to use `range()` to get a list of numbers, you can just apply the `list()` function to it:

```
x = list(range(10))  
print(type(x))  
print(x)
```

In a similar way, you could get a tuple or set of numbers.

Customising range

You can specify a starting value:


```
for i in range(3, 10):  
    print(i)
```

You can specify a step:

```
for i in range(0, 10, 2):  
    print(i)
```

You can work backwards by making the step negative:

```
for i in range(10, 0, -2):  
    print(i)
```

Nesting ranges

You might find yourself using `range` in nested for loops:

```
for i in range(1, 11):  
    for j in range(1, 11):  
        print(f'{i} times {j} is {i*j}')
```

It has become conventional to use `i`, `j`, and `k` as loop counters.

```
for i in range(2):  
    for j in range(2):  
        for k in range(2):  
            print(i, j, k)
```

Files as lists

Just as we can think of a string as a tuple of characters, we can think of a file as **a list of lines**. Python provides two useful methods that take advantage of this.

Readlines

We saw in Week 1 that you can read the content of a text file into a string, using the `read()` method. The `readlines()` method allows you to read the contents into a list of lines instead. This allows you to use a `for` statement to loop through the lines of a file one-by-one.

```
# Add some lines to a file
with open('myfile', 'w') as file:
    file.write('This is the first line\n')
    file.write('This is the second line\n')
    file.write('This is the third line')

# Inspect the results
with open('myfile', 'r') as file:
    lines = file.readlines()
    print(f'The file contains {len(lines)} lines.')
    for line in lines:
        print(f'Line: {line}')
```

Notice the newline character is included in each line. You will typically want to remove that using the `strip()` method of a string.

Writelines

The `writelines()` method lets you write multiple lines to a file, given as a list of strings. Note that each line in the list should be terminated with a newline character (`\n`) otherwise `writelines()` will concatenate the content onto a single line.

```
LINES = [
    'This is the first line\n',
    'This is the second line\n',
    'This is the third line'
]

# Write these lines to a file
with open('myfile', 'w') as file:
    file.writelines(LINES)

# Inspect the results
with open('myfile', 'r') as file:
    print(file.read())
```

Reading CSV files

You might sometimes want to read a CSV file. Python provides a convenient way of reading a CSV file and parsing it into a list of lists. You need to import the `csv` module to use this feature of Python.

```
import csv

# Create a CSV file to experiment with
with open('myfile', 'w') as file:
    file.write('a,b,c\n')
    file.write('d,e,f\n')
    file.write('g,h,i\n')

# Read the CSV file
with open('myfile', 'r') as file:
    lst = list(csv.reader(file))
    print(lst)
```

Dates and times

This is not related to collections, but now is a good time to talk about Python's facilities for working with dates and times.

When working with real-world data you often have to deal with dates and times. Date and times can be tricky to work with, because people format and present them in different ways. Consider, for example, a date written as "02-03-1998". Does this represent 2nd March 1998, or 3rd February 1998? In Australia it would be the former, but in the US it would be the latter.

Python has a `datetime` library that defines `datetime`, `date`, and `time` types. These provide a uniform and comprehensive way to handle dates and times. The `datetime` type is the most flexible and thus the most commonly used. There is also a `timedelta` type, which is used to work with durations (i.e., time intervals).

To use these you need to import them:

```
from datetime import datetime, date, time, timedelta
```

You don't need to import them all - just the ones you will be using.

Usually your goal will be to convert a string or integer representation of a date or time into a `datetime` object, apply whatever processing you need to the object, and then use a formatting function to convert it back to a traditional format.

Creating a datetime object

There are several ways to create a datetime object.

You can directly construct it by providing the year, month, and day, and, optionally, the time and timezone.

```
from datetime import datetime
dt = datetime(
    year = 1968, month = 6, day = 24,
    hour = 5, minute = 30, second = 0
)
print(dt)
```

You can also construct it from a string. In this case you need to tell Python what format the string uses:

```
from datetime import datetime
dt = datetime.strptime('24-06-1968, 05:30:00', '%d-%m-%Y, %H:%M:%S')
print(dt)
```

The trickiest part about this is remembering what the formatting codes are. You'll probably find yourself looking them up quite often. Here are the main ones:

```
%Y Four-digit year (1968)
%y Two-digit year (68)

%B Full month name (June)
%b Abbreviated month name (Jun)
%m Two-digit month number (01-12)

%A Full day name (Monday)
%a Abbreviated day name (Mon)
%d Two-digit day number (01-31)

%H Two-digit hour (00-23)
%I Two-digit hour (00-12)

%M Two-digit minute (00-59)
%S Two-digit second (00-59)

%p AM/PM
```

If you want a datetime object that represents the current date and time you can use the `now` method:

```
from datetime import datetime
dt = datetime.now()
print(dt)
```

Unix timestamp

You can also construct a datetime object from a Unix timestamp. This is one of the most common representations of time. It represents the time as a numerical value - the number of seconds since the Unix epoch, which was at 00:00:00 on Thursday, 1 January 1970 UTC. You can get the current Unix timestamp using the `time.time` function.

```
from time import time
print(time())
```

To create a datetime object from a Unix timestamp you can use use `datetime.fromtimestamp()`.

```
from datetime import datetime
x = datetime.fromtimestamp(1565315907)
print(x)
```

Extracting the components of a datetime object

Once you have a datetime object you can extract its individual components:

```
from datetime import datetime
```

```
dt = datetime(
    year = 1968, month = 6, day = 24,
    hour = 5, minute = 30, second = 0
)
print(dt.year)
print(dt.month)
print(dt.day)
print(dt.hour)
print(dt.minute)
print(dt.second)
```

You can also extract a date object or a time object from a datetime object:

```
from datetime import datetime
dt = datetime(
    year = 1968, month = 6, day = 24,
    hour = 5, minute = 30, second = 0
)
print(dt.date())
print(dt.time())
```

Formatting a datetime object as a string

A very common thing to do is to present a datetime object in a certain format. You can use the `strftime` method, to do this. You need to specify the format you would like, using the same formatting codes as listed above.

```
from datetime import datetime
dt = datetime(
    year = 1968, month = 6, day = 24,
    hour = 5, minute = 30, second = 0
)
print(dt.strftime('%d %B %Y, at %I:%M %p'))
```

Operating on datetime objects

One of the most useful features of datetime objects is that you can very easily calculate new dates and times. The datetime module is so nicely designed that the following examples should be self-explanatory:

First example:

```
from datetime import datetime, timedelta

now = datetime.now()
print(now + timedelta(hours=1))
print(now + timedelta(days=1))
print(now - timedelta(weeks=2))
print(now + timedelta(weeks=4, days=3, hours=2, seconds=45))
```

Second example:

```
from datetime import datetime, timedelta

new_years_eve = datetime(year=2024, month=12, day=31)
now = datetime.now()
time_remaining = new_years_eve - now
print(time_remaining)
```

Third example:

```
from datetime import datetime, timedelta

my_birthday = datetime(year=1988, month=6, day=24)
now = datetime.now()
my_age = now - my_birthday
print(my_age)
```

Further reading

You might find the following helpful:

- The [Python Tutorial](#) at [w3schools.com](#)