Week 2 - Python Part 2

Data structures

A **data structure** is the means by which a program stores pieces of information (i.e. *data*) in a systematic way (i.e. *structured*) so that it can be handled and processed during the program's execution. We have already seen some basic (low-level) data structures with the basic data types (e.g. `ints`, `floats`, `booleans`, `strings`, etc) - these are structured representations of small pieces of information. As programmers, however, we typically consider data structures at a conceptually higher level: using the basic data types as the fundamental units of data and considering structures built up from these - for example *sequences of* `ints`.

This week's content introduces you to some of the most useful built-in data structures provided by Python: lists, tuples, sets and dictionaries. As programming languages have developed, *agglomeration structures* such as these have proven to be the cornerstone to working with data, and Python has some of the most powerful techniques for manipulating these data structures among modern programming languages.

Lists

A list is an ordered container of objects, possibly empty. The objects can be of any type, and they need not all be of the same type. Each is called an *item* of the list.

List literals

You can specify a list using a list literal, which is a sequence of expressions, each of which refers to an object, separated by commas, and surrounded by square brackets. For example:

- `[2, 4, 6, 8]`
- `['cat', 'mouse', 'cat', 'mouse']`
- `[1, 3.14, 'a', True]`
- `[]` (the empty list)

Note that the same object can be included more than once in a list.

Getting the length

We can use Python's `len()` function to find the number of items in a list:

Run PYTHON

```python
print(len([1, 2, 3])) # 3 items
print(len(['12', '45'])) # 2 items
print(len([1, 3.14, 'hello', True])) # 4 items
print(len([])) # 0 items
```

Accessing items

Each item in a list has a unique index, **starting from 0**. So, the first item has index 0, the second item has index 1, and so on.

You can refer to the items in a list by using the indexing operator `[]`:

Run PYTHON

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

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

```python
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. The syntax for slicing is `list[begin:end]`, where `begin` is the start index and `end` is the end index. The `end` index is non-inclusive.

```python
x = ['a', 'b', 'c', 'd', 'e']
print(x[0:3]) # Prints items from index 0 to less than 3
```

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:

```python
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.

```python
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 `list[begin:end:step]`, where `step` is an integer.

```python
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
```

Checking for an item

You can check whether a list contains a certain value by using the `in` keyword:

```python
x = ['a', 'b', 'c', 'd', 'e']
if 'c' in x:
    print("'c' is in the list")
else:
    print("'c' is not in the list")
```

Changing items

You can change the value of an item by referring to it using the indexing operator and then assigning it a value:

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

Reversing items

You can reverse the order of items in a list by using the `reverse()` method:

```python
x = ['dog', 'cat', 'mouse', 'horse', 'goat']
x.reverse()
print(x)
```

Adding items

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

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

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

```python
x = ['a', 'b', 'c', 'd', 'e']
x.insert(2, 'x')
print(x)
```

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

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

You can join two lists by using the `+` operator:

```python
x = ['a', 'b', 'c', 'd', 'e']
y = ['f', 'g', 'h']
print(x + y)
```

Removing items

You can remove an item by index by using the `pop()` method, if `pop()` isn't given an index then the last index will be removed, the removed element will be returned:

```python
x = ['a', 'b', 'c', 'd', 'e']
print(x.pop(2))
print(x)
print(x.pop())
print(x)
```

You can remove any item by its index with the `del` keyword:

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

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

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

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

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

Other methods

You can use the `count()` method to find the number of times a specified value appears in a list:

▸ Run PYTHON

```python
x = [1, 3, 7, 8, 7, 5, 4, 6, 8, 5]
print(x.count(7))
```

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

▸ Run PYTHON

```python
x = [1, 3, 7, 8, 7, 5, 4, 6, 8, 5]
print(x.index(7))
```

Other functions

`sum()` computes the sum of a list:

▸ Run PYTHON

```python
print(sum([1, 3, 7, 8, 7, 5, 4, 6, 8, 5]))
```

`all()` returns `True` if `all` items evaluate to `True` using the `bool()` function.

▸ Run PYTHON

```python
print(all([True, True, True]))
print(all(['a', 'a', 'a']))
print(all(['', 'a', 'b']))
```

`any()` returns `True` if `any` items evaluate to `True` using the `bool()` function.

▸ Run PYTHON

```python
print(any([False, True, True]))
print(any(['a', 'a', 'a']))
print(any(['', 'a', 'b']))
print(any([False]))
```

For functional programming python3 also has `map()`, `reduce()`, and `filter()` functions.

Sorting lists

Basic sorting

You can sort the elements of a list by using the `sort()` method, which orders the items by comparing their values using the `<` operator.

```python
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 reverse order by passing the keyword argument `reverse=True:`

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

You can also sort the list by using the function `sorted`:

```python
x = [1, 5, 4, 2, 3]

y = sorted(x)
print(y)

y = sorted(x, reverse=True)
print(y)
```

Sorting with a function

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

What you need to do is sort using a function that returns, for each item in the list, a value that you would like to sort the item by. In this case we want to sort by length, so we can use the len() function.

▸ Run PYTHON

```python
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.

Looping through lists

For loops

It is very common to loop through the items in a list one by one. You could access each element using its index - and use a `while` statement to go through the whole list:

▶ Run PYTHON

```python
vowels = ['a', 'e', 'i', 'o', 'u']
i = 0
while i < len(vowels):
    print(vowels[i])
    i += 1
```

But it is better to use a `for` loop, which is custom-made for this kind of thing:

▶ Run PYTHON

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

Break and continue

`break` and `continue` can also be used inside `for` loops.

▶ Run PYTHON

```python
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)
```

Enumerating

Sometimes when we loop through a list we need to use the index of an item as well as its value. We can do this by using the `enumerate()` function.

▸ Run PYTHON

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

Range

You might find yourself wanting to loop through a sequence of numbers. The `range()` function is a good way to do this:

Run PYTHON

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

`range(n)` returns the numbers 0, ..., n-1. Actually, it doesn't actually return the numbers themselves, but a method for generating each number as it is required.

You can also specify a starting value:

Run PYTHON

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

And you can specify a step:

Run PYTHON

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

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

Run PYTHON

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

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

Run PYTHON

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

List of lists

The items in a list can be of any type. In particular, they can be lists. So it's possible to create *lists of lists*. They behave just like any other list.

▸ Run PYTHON

```python
numbers = [[1,2,3],[3,4,5],[6,7,8]]
print(len(numbers))
print(numbers[0])

numbers.reverse()
print(numbers)
```

Notice that `numbers` has length 3 because it contains just 3 items (each of which also happens to be a list of items). Notice that `numbers[0]` is the first of the three items, which happens to be a list. Notice that the `reverse()` method reverses the order of the 3 items, but doesn't do anything to the items themselves.

Since `numbers[0]` is a list, we can do all of the usual lists things with it:

▸ Run PYTHON

```python
numbers = [[1,2,3],[3,4,5],[5,6,7]]
print(len(numbers[0]))
print(numbers[0][0])
numbers[0].reverse()
print(numbers[0])
```

When we have a list of lists we might end up using *nested for loops*. Suppose that we want to add up all of the numbers that appear in `numbers`. We can do this as follows:

▸ Run PYTHON

```python
numbers = [[1,2,3],[3,4,5],[5,6,7]]
total = 0
for i in numbers:
    for j in i:
        total += j
print(total)
```

Strings as lists

In Python we can think of a string as being a list of characters.

Accessing characters

We can access characters using indices, and we can get slices of strings. But we cannot reassign individual elements.

Indexing on strings

Like lists, each character inside a string can be accessed using the `[]` indexing operator.

```python
x = 'abcd'
print(x[1])
```

However, the result of the `[]` indexing operator always returns a string. Even on strings with a single element.

```python
x = 'a'
print(x[0])
print(x[0] == x)
print(type(x[0]))
```

Fun fact: this means that we can read the zeroth element repeatedly, this has no practical value and is just a side effect of the way string indexing works in Python.

```python
x = 'a'
print(x[0][0][0][0][0][0][0][0][0][0][0][0][0])
```

The other major difference between strings and lists is that the values on a particular index cannot be reassigned:

```python
# Strings CANNOT be reassigned using the index
x = 'abcd'
x[1] = 'X'
print(x)
```

Slicing strings

The syntax we've seen for slicing lists will also work for strings:

```python
y = 'abcdef'
print(y[1:3])
print(y[1:-2])
```

Splitting strings

One of the most useful methods available for string objects is the `split()` method. This will break a string up into a list of substrings that were separated by a delimiter. Note the delimiter is removed from the list of substrings.

```python
y = 'The cat sat on the mat'
print(y.split(' '))

z = '12:30:45'
print(z.split(':'))
```

Files as lists

Python provides two useful methods for reading and writing multiple lines of text (given as lists of strings).

The `readlines()` method

We saw in Week 1 that we can either read the content of a text file one line at a time using the `readline()` method, or all at once using the read() method. The `readlines()` method is a compromise between the two approaches: reading all the content of a file, but returning a line-by-line list of the results.

```python
# Set up the file for reading
with open('myfile', 'w') as file:
    file.write('Line 1: Some content\n')
    file.write('Line 2: ...\n')
    file.write('Line 3: Last line')

with open('myfile', 'r') as file:
    content = file.readlines()

print(content) # Note that the newline characters are included in the li
```

The `writelines()` method

The `writelines()` method lets you write multiple lines (given as a list of strings) to a file. 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.

```python
lines = ['Line 1: The cat\n', 'Line 2: sat on\n', 'Line 3:', ' the mat']

with open('myfile', 'w') as file:
    file.writelines(lines)

# Open the file to see the result
with open('myfile', 'r') as file:
    content = file.readlines()
    for line in content:
        print(line)
```

Files as lists

Because processing files one line at a time is a very common task, Python allows you to use the `for` command to iterate through the lines of a file as if a file object were a list of lines.

```python
# Set up the file for reading
with open('myfile', 'w') as file:
    line_one = 'Line 1: Some content\n'
    line_two = 'Line 2: ...\n'
    line_three = 'Line 3: Last line'
    file.writelines([line_one, line_two, line_three])

# Demonstrate using the for command to loop through the lines of a file
with open('myfile', 'r') as file:
    for line in file:
        print(line)
        print(line.endswith('\n'))

```

Tuples

A tuple is a list that is *immutable*, which means that it cannot be modified.

Tuple literals

Tuple literals are like list literals, except you use round brackets rather than square brackets:

- (2, 4, 6, 8)
- ('cat', 'mouse', dog')
- (1, 3.14, 'a', True)
- () (the empty tuple)

Immutability

You can mostly work with tuples just like you work with lists:

- Use `len()` to find the number of items

```python
x = ('a', 'b', 'c', 'd', 'e')
y = (1, 1, 2, 3)
print(len(x))
print(len(y))
```

- Use `in` to check for an item and `count` to count the number of occurrences of an item

```python
y = (1, 1, 2, 3, 5, 8)
print(4 in y)
print(y.count(1))
```

- Use `[]` to refer to items of the list, including slicing

```python
x = ('a', 'b', 'c', 'd', 'e')
print(x[2])
print(x[-1])
print(x[1:3]) # Note this will also give a tuple
```

- Join two tuples with `+`

```python
x = (1, 2, 3)
y = (4, 5, 6)
z = x + y # Note this will give a tuple
print(z)
```

- Use `for` to loop through the items
- Use `sum`, `all`, and `any` to combine items of the appropriate type.

```python
y = (1, 1, 2, 3)
z = (True, False, True)
print(sum(y))
print(all(z))
print(any(z))
```

But because tuples are **immutable** there are various things that you cannot do:

- Change the elements of the tuple:

```python
x = (0, 2, 3)
x[0] = 1
```

- Add items to the tuple:

```python
x = (1, 2, 3)
x.append(4)
```

- Remove elements from the tuple:

```python
x = (1, 2, 3)
x.remove(2)
```

- Sort the tuple:

```python
x = (1, 3, 2)
x.sort()
```

Usage

Tuples can be viewed as multi-dimensional literals, for example a two-dimensional point `(1,3)`, a name-surname pairing: `('John', 'Smith')`, or a day-month-year triple representing a date `(26,1,2020)`. The immutability of tuples binds the individual components tightly together - it is not possible for a program to alter one component separately from the others and the *order* of the elements is important: `('John', 'Smith')` and `('Smith', 'John')` are fundamentally different objects. So it is more sensible to view a tuple as a single, complex, object, rather than as a collection of smaller objects. This makes tuples ideal for representing data where there is a strong connection between the components and where it should be difficult for a programmer to alter one aspect independently of the others.

While a list can almost always be used where a tuple can be, by using tuples in the appropriate setting, you are signalling to anyone reviewing your code that the *guarantee of immutability* of the object is important.

Sets

A set is an *unordered* container of *unique* objects (i.e. the same object cannot be included more than once).

Set literals

Set literals are like list literals, except you use curly brackets rather than square brackets:

- {2, 4, 6, 8}
- {'cat', 'mouse', dog'}
- {1, 3.14, 'a', True}
- {} (the empty set)

Unordered and unique

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 square brackets notation you will get an error:

Run PYTHON

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

Because they are not ordered you can't be sure what order Python will list them in (try running the following snippet multiple times):

Run PYTHON

```python
vowels = {'a', 'e', 'i', 'o', 'u'}
print(vowels)
```

('Smith', 'John')('Smith', 'John')('Smith', 'John')('Smith', 'John')('Smith', 'John')

The items are also *unique*, which means that the same object cannot be included more than once in the set. If you try to include the same object more than once Python will ignore all but the first:

Run PYTHON

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

Counting items

You can use Python's `len()` function to get the number of items in a set:

▸ Run PYTHON

```python
print(len({'a', 'e', 'i', 'o', 'u'}))
print(len({}))
```

Checking for an item

You can use the `in` keyword word to check whether an item is in a set:

▸ Run PYTHON

```python
names = {'John', 'Bob', 'Alice'}
print('John' in names)
print('Eve' in names)
```

Looping through items

You can loop through the items in a set by using a `for` loop:

▸ Run PYTHON

```python
for vowel in {'a', 'e', 'i', 'o', 'u'}:
    print(vowel.upper())
```

⚠ Because sets are unordered, there is no guarantee in what order the items will be iterated through.

Adding items

You can add a single item to a set by using the set's `add()` method:

▸ Run PYTHON

```python
vowels = set()
vowels.add('a')
vowels.add('e')
vowels.add('e') # No error, just not added
print(vowels)
```

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

▸ Run PYTHON

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

Removing items

You can remove items from a set 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:

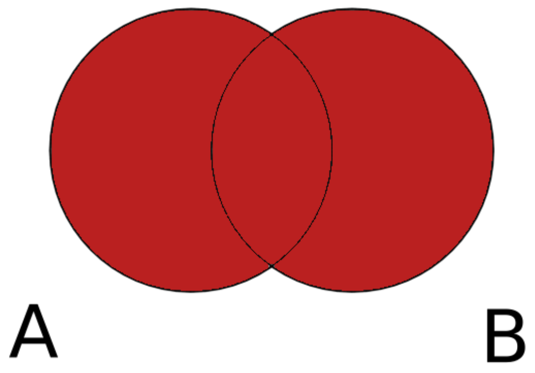
```python
vowels = {'a', 'e', 'i', 'o', 'u'}
vowels.remove('a')
print(vowels)
vowels.discard('f') # No error
print(vowels)
vowels.remove('f') # Error
```

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

```python
vowels = {'a', 'e', 'i', 'o', 'u'}
vowels.clear()
print(vowels)
```

Union of two sets

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

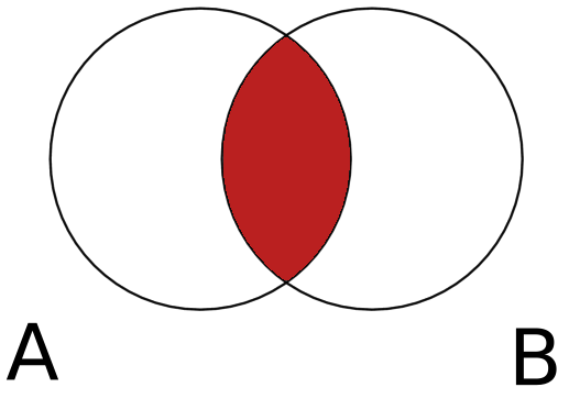


You can get the union of two sets using either the `|` operator or the `union()` method:

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

Intersection of two sets

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



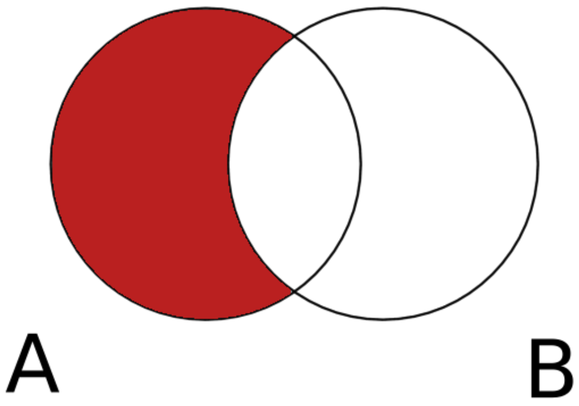
You can get the intersection of two sets using either the `&` operator or the `intersection()` method:

Run PYTHON

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

Difference between two sets

The difference between set A and set B is the set of items in A but not B:



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

Run PYTHON

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

Usage

Just as with tuples, a list can often be used where a set is being used, raising the question, "what are the advantages of using sets rather than lists?" Besides the set operations `union`, `intersection`, and `difference` available only to sets, the primary reason to use the set structure over the list structure is that it *guarantees* the uniqueness of elements and unordered storage. While there may be some theoretical efficiency to be gained in terms of how Python stores sets rather than lists, the primary reason for using the set structure from a programming perspective should be to signal to anyone reviewing your code that these properties are important.

Dictionaries

A dictionary is an unordered collection of key-value pairs. e.g. `{'name': 'John', 'age': 54}`. It represents a mapping from a set of keys to a set of values.

Dictionary literals

Dictionary literals are presented in curly braces like set literals, with key-value pairs separated by commas, and each key and value separated with a colon.

Run PYTHON

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

A key and a value can be any object. Keys do not all have to be the same type, and neither do values; however it is good programming practice to keep the types consistent.

Run PYTHON

```python
dictionary = {
    1: 'one',
    'two': 2, # Mixing key types is not good practice as it can lead to
    3: 3
}

print(dictionary)
```

Uniqueness of keys

In a dictionary, values can be duplicated, but keys must be unique. If multiple items with the same key are given, then only the latest value is stored.

Run PYTHON

```python
scores = {
    'Alice': 1,
    'Bob': 1
}

print(scores)

scores = {
    'Alice': 0,
    'Alice': 1
}

print(scores)
```

Indexing

Items can be retrieved from the dictionary using the indexing operator. Note that indices of a dictionary are key values, which are not necessarily integers.

Run PYTHON

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

Updating

Updating an item in a dictionary is the same as updating an item in a list.

Run PYTHON

```python
scores = {
    'Alice': 0,
    'Bob': 1,
    'Eve': 2,
    'Mallory': 3,
}
scores['Bob'] = 900
scores['Alice'] += 1
print(scores)
```

Deletion

The `del` keyword works the same on dictionaries as it does on lists.

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

Iteration

Iterating a dictionary directly yields only the keys.

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

Fortunately Python dictionaries provide an easy access function, `items()`, to get the items as well:

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

Usage

While lists, tuples and sets represent collections of objects, dictionaries represent a *relationship* between two collections: a set of keys and a set of values. The relationship is a many-one relationship - multiple keys can be related to the same value, but a single key can only be related to at most one value. This is not too much of a restriction - a reasonably simple way to overcome this if there is a need to map a key to multiple values is to use a list (or set) of items as the value in the key-value pairs.

A situation that commonly arises, particularly when naively programming with data, is an abundance of lists that all need to align on their indices: for example, the third item in list x corresponds to the third item in list y. If you find yourself in this situation, then you should be working with dictionaries. Dictionaries are a robust and safe way of managing relationships between data items.

It may appear that lists can be viewed as dictionaries where the list indices are the keys, however there are some notable differences - particularly after items have been removed. In the case of lists, the indices of the items appearing after a removed item will all be changed, whereas for a dictionary the keys are not affected when an item is removed. Each effect has its benefits: from a list perspective, the "next available key" is easy to find, resulting in minimal overhead for adding new items; whereas from a dictionary point of view there is stability between the item and the index it is assigned, as well as a (partial) record of the deletions that have occurred.

Converting between lists, tuples, sets and dictionaries

Like the `int()`, `float()`, `str()` type conversion functions, the `list()`, `tuple()`, `set()` functions facilitate the conversion between these three container types.

```python
x = [1, 1, 1, 2, 3, 4, 5]
print(tuple(x))
print(set(x))

x = { 1, 2, 3 }
print(list(x))
print(tuple(x))

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

The `list()` function can also be applied to dictionaries, but it will only return the collection of keys. To obtain the full set of key-value pairs, call the `list()` function on the result of the dictionary's `items()` method

```python
scores = {
    'Alice': 0,
    'Bob': 1,
    'Eve': 2,
    'Mallory': 3,
}

# Using the list() function will yield only keys
print(list(scores))

# Using the list() function on .items will yield the tuples
print(list(scores.items()))
```

The dict() function will convert a list of key-value pairs into a dictionary.

Run PYTHON

```python
# A list of tuples can also be converted to a dictionary
scores = [(1, 'A'), (2, 'B')]
scores_dict = dict(scores)

print(type(scores))
print(type(scores_dict))
```

List comprehension

Consider the following, reasonably common snippet of code, that builds a new list/set/dictionary from an existing one:

```python
x = range(5)
y_list = []
y_set = set()
y_dict = {}

for z in x:
    y_list.append(z*z-z)
    y_set.add(z*z-z)
    y_dict[z] = z*z-z

print(y_list)
print(y_set)
print(y_dict)
```

Python provides a compact way of constructing lists, sets, and dictionaries from existing enumerable structures via **list comprehension**.

```python
x = range(5)

y_list = [z*z-z for z in x]
y_set = {z*z-z for z in x}
y_dict = {z: z*z-z for z in x}

print(y_list)
print(y_set)
print(y_dict)
```

Further reading

You might find the following helpful:

- The Python Tutorial at w3schools.com

Practice quiz

The following quiz will test your Python knowledge with respect to Week 1 and Week 2 contents:

- The Python Quiz from w3schools.com