Ch.6: Dictionaries and strings

Joakim Sundnes[1,2]

[1]Simula Research Laboratory
[2]University of Oslo, Dept. of Informatics

Oct 9, 2019

In this chapter we will mainly focus on two data types; dictionaries and strings. Dictionaries may be seen as a generalization of the list datatype, where the indices are not required to be integers. We have already used strings in all previous chapters, but we will revisit them here to introduce a number of new and useful functions. Both dictionaries and strings are particularly useful for reading and processing text files, and many of our examples will be related to this application.

1 Implementing mappings in Python

In mathematics, a mapping is a relationship between objects or structures, which often takes the form of a function. A mapping f is a rule that assigns a unique value $f(x)$ to a given input x. Mappings are also widely used in computer science, and can be implemented in many different ways. For instance, a Python list may be viewed as a mapping between integers (list indices) and the objects contained in the list. More general mappings can be implemented using functions and if-tests, for instance the mapping

```
'Norway' --> 'Oslo'
'Sweden' --> 'Stockholm'
'France' --> 'Paris'
```

could be implemented in Python as

```
def f(x):
    if x == 'Norway':
       return 'Oslo'
    elif x == 'Sweden':
       return 'Stockholm'
    elif x == 'France':
       return 'Paris'
```

Such an implementation is obviously not very convenient if we have a large number of input- and output-values. An alternative implementation of the mapping would be to use two lists of equal length, where for instance item n in

list `countries` correponds to item n in list `capitals`. However, since such general mappings are useful in many context, Python provides a special datastructure for them called a *dictionary*. Data structures similar to a dictionary are used in many programming languages, but often have different names. Commonly used names are associative array, symbol table, hashmap, or simply a map.

A dictionary may be seen as a generalization of a list, where the indices are not required to be integers, but can be any immutable Python data type. The "indices" of a dictionary are called *keys*, and in this course we will typically use strings as dictionary keys. The dictionary implementation of the mapping above looks like

```
d = {'Norway':'Oslo','Sweden':'Stockholm','France':'Paris'}
```

and we can look up values in the dictionary just as we would in a list, using the dictionary *key* instead of an index:

```
print(d['Norway'])
```

To extend the dictionary with new values, we can simply write

```
d['Germany'] = Berlin
```

Notice this important difference between a list and a dictionary. For a list we had to use `append()` to add new elements. A dictionary has no `append` method, and to extend it we simply introduce a new key and corresponding value.

Dictionaries can be initialized in two different ways. One is by using the curly brackets, as in the example above. Alternatively, we can use the built-in function `dict`, which takes a number of key-value pairs as arguments and returns the corresponding dictionary. The two approaches may look like this:

```
mydict = {'key1': value1, 'key2': value2, ...}

temps = {'Oslo': 13, 'London': 15.4, 'Paris': 17.5}

# or
mydict = dict(key1=value1, key2=value2, ...)

temps = dict(Oslo=13, London=15.4, Paris=17.5)
```

Notice the differences in syntax. When initializing using the curly brackets we use colon to separate the key from its corresponding value, and the key can be any immutable Python object (strings in the example above). When using the `dict` function, we pass the key-value pairs as *keyword arguments* to the function, and the keywords are converted to keys of type string. However, in both cases the initialization involves defining a set of key-value pairs to populate the dictionary. A dictionary is simply an unordered collection of such key-value pairs.

We are used to looping over lists to access the individual elements. We can do the same with dictionaries, with the small but important difference that looping over a dictionary means looping over the keys, not the values. If we want to access the values we need to look them up in the dictionary using the keys. For instance, generic code to print all the values of a dictionary would look as follows:

```
for key in dictionary:
    value = dictionary[key]
    print(value)
```

A conrete example based on the example above may look like

```
temps = {'Oslo': 13, 'London': 15.4, 'Paris': 17.5, 'Madrid': 26}
for city in temps:
    print(f'The {city} temperature is temps{city}')
```

Output:

```
The Paris temperature is 17.5
The Oslo temperature is 13
The London temperature is 15.4
The Madrid temperature is 26
```

As mentioned above, a dictionary is an *unordered* collection of key-value pair, meaning that the sequence of the keys in the dictionary is arbitrary. If we want to print or otherwise process the elements in a particular order the keys first need to be sorted, for instance using the builtin function `sorted`:

```
for city in sorted(temps):   # alphabetic sort of keys
    value = temps[city]
    print value
```

There may be applications where sorting the keys like this is important, but usually the order of a dictionary is insignificant. In most applications where the order of the elements is important, a list or an array is a more convenient data type than a dictionary.

Dictionaries and lists share many similarities. Much of the functionality that we are familiar with for list also exists for dictionaries. We can, for instance, check if a dictionary has a particular key with the expression `key in dict`, which returns True or False:

```
>>> if 'Berlin' in temps:
...     print('Berlin:', temps['Berlin'])
... else:
...     print('No temperature data for Berlin')
...
No temperature data for Berlin
>>> 'Oslo' in temps     # standard boolean expression
True
```

Deleting an element of a dictionary is done exactly as with lists, using the operator `del`:

```
>>> del temps['Oslo']   # remove Oslo key w/value
>>> temps
{'Paris': 17.5, 'London': 15.4, 'Madrid': 26.0}
>>> len(temps)          # no of key-value pairs in dict.
3
```

In some cases it may be useful to access the keys or values of a dictionary as separate entities, and this can be obtained with the methods `keys` and `values`,

for instance `temps.keys()` and `temps.values()` for the case above. These methods will return *iterators*, which are list-like objects that can be looped over or converted to a list:

```
>>> for temp in temps.values():
>>>     print(temp)
...
17.5
15.4
26.0
>>> keys_list = list(temps.keys())
```

Just as with lists, when we assign an existing dictionary to a new variable, the dictionary is not copied. Instead, the new variable name becomes a *reference* to the same dictionary, and changing it will also change the original variable. The following code illustrates the behavior:

```
>>> t1 = temps
>>> t1['Stockholm'] = 10.0     # change t1
>>> temps                      # temps is also changed!
{'Stockholm': 10.0, 'Paris': 17.5, 'London': 15.4,
  'Madrid': 26.0}
>>> t2 = temps.copy()          # take a copy
>>> t2['Paris'] = 16
>>> t1['Paris']                # t1 was not changed
17.5
```

Here, the call to `temps.copy()` ensures `t2` is a copy of the original dictionary, and not a reference, so changing it does not alter the original dictionary. Recall that lists behave in the same way:

```
>>> L = [1, 2, 3]
>>> M = L
>>> M[1] = 8
>>> L[1]
8
>>> M = L.copy() #for lists, M = L[:]  also works
>>> M[2] = 0
>>> L[2]
3
```

So far we have used texts (string objects) as keys, but the keys of a dictionary can be any *immutable* (constant) object. For instance, we can use integers, floats, and tuples as keys, but not lists since they are mutable objects:

```
>>> d = {1: 34, 2: 67, 3: 0}   # key is int
>>> d = {13: 'Oslo', 15.4: 'London'} # possible
>>> d = {(0,0): 4, (1,-1): 5}  # key is tuple
>>> d = {[0,0]: 4, [-1,1]: 5}  # list is mutable/changeable
...
TypeError: unhashable type: 'list'
```

Of course, the fact that these alternatives work in Python does not mean that they are recommended or very useful. It is, for instance, hard to imagine a useful application for a dictionary with a temperature as key and a city name as value. Strings are the most obvious and commonly used data type for dictionary keys, and will also be the most common through this course. However, there are applications where other types of keys are useful, as we will see in the following examples.

Example; representing a polynomial with a dictionary. The information in the polynomial

$$p(x) = -1 + x^2 + 3x^7$$

can be represented by a dictionary with power as key (`int`) and coefficient as value (`float` or `int`):

```
p = {0: -1, 2: 1, 7: 3}
```

More generally, a polynomial written on the form

$$p(x) = \sum_{i \in I}^{N} c_i x^i,$$

for some set of integers I, can represented by a dictionary with keys i and values c_i. To evaluate a polynomial represented by such a dictionary, we need to iterate over keys dictionary, extract the corresponding values, and sum up the terms. The following function takes two arguments; a dictionary `poly` and a number or array `x`, and evaluates the polynomial in `x`: number (or array) x:

```
def eval_poly_dict(poly, x):
    sum = 0.0
    for power in poly:
        sum += poly[power]*x**power
    return sum
```

We see that the function follows our standard recipe for evaluating a sum; set a summation variable to zero and then add in all the terms using a for-loop. We can write an even shorter version of the function using Python's builtin function `sum`:

```
def eval_poly_dict(poly, x):
    # Python's sum can add elements of an iterator
    return sum(poly[power]*x**power for power in poly)
```

Since the keys of the polynomial dictionary are integers, we could also replace the dictionary with a list, where the list index corresponds to the power of the respective term. The polynomial above, i.e. $-1 + x^2 + 3x^7$ can be represented as the list

```
p = [-1, 0, 1, 0, 0, 0, 0, 3]
```

and the general polynomial $\sum_{i=0}^{N} c_i x^i$ is stored as `[c0, c1, c2, ..., cN]`. The function to evaluate a polynomial represented by a list is nearly identical to the function for the dictionary. The function

```
def eval_poly_list(poly, x):
    sum = 0
    for power in range(len(poly)):
        sum += poly[power]*x**power
    return sum
```

will evaluate a polynomial $\sum_{i=0}^{N} c_i x^i$ for a given x. Although the two representations are very similar, the list representation has the obvious disadvantage that we need to store all the zeros. For "sparse" polynomials of high order this can be quite inconvenient, and the dictionary representation is obviously better. The dictionary representation can also easily handle negative powers, for instance $\frac{1}{2}x^{-3} + 2x^4$:

```
p = {-3: 0.5, 4: 2}
print eval_poly_dict(p, x=4)
```

This code will work just fine without any modifications of the `eval_poly_dict` function. Lists in Python cannot have negative indices (since indexing a list with a negative number implies counting indices from the end of the list), and it is not trivial to extend the list representation to handle negative powers.

Example; read file data into a dictionary. Say we have a file `deg2.dat`, containing temperature data for a number of cities:

```
Oslo:          21.8
London:        18.1
Berlin:        19
Paris:         23
Rome:          26
Helsinki:      17.8
```

We now want to read this file and store the information in a dictionary, with the city names as keys and the temperatures as values. The recipe is nearly identical to the one we previously used for reading file data into lists; first create an empty dictionary, then fill it with values read from the file:

```
with open('deg2.dat', 'r') as infile:
    temps = {}                  # start with empty dict
    for line in infile:
        city, temp = line.split()
        city = city[:-1]        # remove last char (:)
        temps[city]  = float(temp)
```

The only real difference between this code and previous examples based on lists is the way we add new data to the dictionary. We used the the `append` method to populate an empty list, but dictionaries have no such method. Instead, we add a new key-value pair with the line `temps[city] = float(temp)`. Apart from this technical difference, the recipe for populating a dictionary is exactly the same as for lists.

2 String manipulation

We have already worked with strings in previous chapters, for instance the very useful `split`-method:

```
>>> s = 'This is a string'
>>> s.split()
['This', 'is', 'a', 'string']
```

String manipulation is essential for reading and interpreting the content of files, and the way we process files is often quite dependent on the file structure. For instance, we need to know on which line the relevant information starts, how data items are separated, and how many data items there are on each line. The algorithm for reading and processing the text needs to be tailored to the structure of the file. Although the `split` function already considered is quite flexible, and covers most of our needs in this course, it may not always be the best tool. Python has a number of other ways to process strings, which may in some cases make the text processing easier and more efficient.

Text in Python is represented by string objects (of type `str`). To introduce some of the basic operations on strings, we consider the example string:

```
>>> s = 'Berlin: 18.4 C at 4 pm'
```

Such a string is really just a sequence of characters, and it behaves much like other sequence data types such as lists and tuples. For instance, we can index a string to extract individual characters;

```
>>> s[0]
'B'
>>> s[1]
'e'
>>> s[-1]
'm'
```

Slices also work as we are used to, and can be used to extract substrings of a string:

```
>>> s
'Berlin: 18.4 C at 4 pm'
>>> s[8:]      # from index 8 to the end of the string
'18.4 C at 4 pm'
>>> s[8:12]    # index 8, 9, 10 and 11 (not 12!)
'18.4'
>>> s[8:-1]
'18.4 C at 4 p'
>>> s[8:-8]
'18.4 C'
```

Iterating over a string also works as we would expect:

```
>>> s = 'Berlin: 18.4 C at 4 pm'
>>> for s_ in s:
            print(s_, end=' ')
```

Strings have a method named `find`, which searches a string for a given substring, and returns the index of its location:

```
>>> s.find('Berlin')  # where does 'Berlin' start?
0                     # at index 0
>>> s.find('pm')
20
>>> s.find('Oslo')    # not found
-1
```

Lists do not have `find`-method, but we have seen previously seen the list's `index` method, which is quite similar. Strings also have a method named `index`, which does almost the same thing as `find`. However, while `find` will return -1 if the substring does not exist in the string, `index` will end with an error message. If we want to know if a substring is part of a string, and don't really care about its location, we can also use `in`:

```
>>> 'Berlin' in s:
True
>>> 'Oslo' in s:
False

>>> if 'C' in s:
...     print 'C found'
... else:
...     print 'no C'
...
C found
```

We may recall from Chapter 2 that this use of `in` is almost exactly the same as for lists and tuples. The only minor difference is that for lists and tuples we can only check for the existence of an individual element, while for strings it works for a substring of arbitrary length.

In many cases we are interested not only in finding a substring, but to find it and replace it with something else. For this task we have a string method named `replace`. It takes two strings as arguments, and a call like `s.replace(s1, s2)` will replace `s1` by `s2` everywhere in `s`. The following examples illustrate how it is used;

```
>>> s = 'Berlin: 18.4 C at 4 pm'
>>> s.replace(' ', '__')
'Berlin:__18.4__C__at__4__pm'
>>> s.replace('Berlin', 'Bonn')
'Bonn: 18.4 C at 4 pm'
>>> s.replace(s[:s.find(':')], 'Bonn')
'Bonn: 18.4 C at 4 pm'
```

In the final example we combine `find` and replace to replace all text before the `':'` with `'Bonn'`. First, `s.find(':')` returns 6, which is the index where the `':'` is found, then the slice `s[:6]` is `'Berlin'`, which is replaced by `'Bonn'`.

Splitting and joining strings. We have already introduced the `split` method, which is arguably the most useful method for reading and processing text files. As we recall from Chapter 4, the call `s.split(sep)` will split the string `s` into a list of substrings separated by `sep`. The `sep` argument is optional, and if it is omitted the string is split with respect to whitespace. Consider these two simple examples to recall how it is used;

```
>>> s = 'Berlin: 18.4 C at 4 pm'
>>> s.split(':')
['Berlin', ' 18.4 C at 4 pm']
>>> s.split()
['Berlin:', '18.4', 'C', 'at', '4', 'pm']
```

The `split` method has an "inverse" called `join`, which is used to put a list of strings together with a delimiter in between:

```
>>> strings = ['Newton', 'Secant', 'Bisection']
>>> ', '.join(strings)
'Newton, Secant, Bisection'
```

Notice that we call the `join` method belonging to the delimiter `', '`, which is a string object, and pass the list of strings as argument. If we want to put the same list together separated by whitespace, we would simply replace `', '.join(strings)` in the example above with `' '.join(strings)`.

Since `split` and `join` are inverse operations, using them in sequence will give back the original string, as in the following example;

```
>>> l1 = 'Oslo: 8.4 C at 5 pm'
>>> words = l1.split()
>>> l2 = ' '.join(words)
>>> l1 == l2
True
```

A common usecase for the join method is to split off a known number of words on a line. Say we want to read a file on the following format, and combine the city name and the country into a single string;

```
Tromsø Norway 69.6351 18.9920 52436
Molde Norway 62.7483 7.1833 18594
Oslo Norway 59.9167 10.7500 835000
Stockholm Sweden 59.3508 18.0973 1264000
Uppsala Sweden 59.8601 17.6400 133117
```

The following code will read such a file and create a dictionary with the data

```
cities = {}
with open('cities.txt') as infile:
    for line in infile:
        words = line.split()
        name = ', '.join(words[:2])
        data = {'lat': float(words[2]),  'long':float(words[3])}
        data['pop'] = int(words[3])
        cities[name] = data
```

Here the line `name = ', '.join(words[:2])` creates strings like `'Tromsø, Norway'`, which are then used as dictionary keys.

In most of the examples considered so far we have read and processed text files line by line, but in some cases we have a string with lots of text in multiple lines, and we want to split it into single lines. We can do this using the `split` method with the appropriate separator. For instance, on Linux and Mac systems the line separator is `\n`;

```
>>> t = '1st line\n2nd line\n3rd line'
>>> print t
1st line
2nd line
3rd line
>>> t.split('\n')
['1st line', '2nd line', '3rd line']
```

This example works fine on Mac or Linux, but the line separator on Windows is not `\n`, but `\r\n`, and to have a platform independent solution it is better to use the method `splitlines()`, which works with both line separators;

```
>>> t = '1st line\n2nd line\n3rd line'           #Unix format
>>> t.splitlines()
['1st line', '2nd line', '3rd line']
>>> t = '1st line\r\n2nd line\r\n3rd line' # Windows
>>> t.splitlines()                        # cross platform!
['1st line', '2nd line', '3rd line']
```

Strings are constant - immutable - objects. In many of the examples above we have highlighted the similarity between strings and lists, since we are very familiar with lists from earlier chapters. However, strings are even more similar to tuples, since they are immutable objects. We could change elements of a list in-place by indexing into the list, but this does not work for strings. Trying to assign a new value to a part of a string will result in an error message:

```
>>> s[18] = 5
...
TypeError: 'str' object does not support item assignment
```

Instead, to perform such a replacement we can build a new string manually by adding pieces of the original string, or use the `replace` method introduced above:

```
>>> # build a new string by adding pieces of s:
>>> s2 = s[:18] + '5' + s[19:]
>>> s2
'Berlin: 18.4 C at 5 pm'
>>> s2 = s.replace(s[18],5)
>>> s2
'Berlin: 18.4 C at 5 pm'
```

Some may find it confusing that strings are immutable, but they still have a method like `replace`, which apparently alters the string. How can we replace a substring with another if strings are immutable objects? The answer is that `replace` does not really change the original string, but returns a new one. This behavior is similar to for instance the call `s.split()`, which will not turn `s` into a list but instead leave `s` unchanged and *return* a list of the substrings. Similarly, a call like `s.replace(4,5)` does not change `s` but it will return a new string that we can assign either to `s` or some other variable name, as we did in the example above. The call `s.replace(4,5)` does nothing useful on its own, unless it is combined into an assignment such as `s2 = s.replace(4,5)` or `s = s.replace(4,5)`.

Other convenient string methods in Python. It is often convenient to strip off leading or trailing whitespace from a string, and there are methods `strip()`, `lstrip()` and `rstrip()` for doing this:

```
>>> s = '   text with leading/trailing space   \n'
>>> s.strip()
```

```
'text with leading/trailing space'
>>> s.lstrip()   # left strip
'text with leading/trailing space   \n'
>>> s.rstrip()   # right strip
'   text with leading/trailing space'
```

We can also check whether a string is only contains numbers (digits), only space, or if a string starts or ends with a given substring;

```
>>> '214'.isdigit()
True
>>> '  214 '.isdigit()
False
>>> '2.14'.isdigit()
False

>>> '    '.isspace()   # blanks
True
>>> '  \n'.isspace()   # newline
True
>>> '  \t '.isspace()  # TAB
True
>>> ''.isspace()       # empty string
False

>>> s.startswith('Berlin')
True
>>> s.endswith('am')
False
```

Finally, we may be interested in converting between lower case and upper case characters;

```
>>> s.lower()
'berlin: 18.4 c at 4 pm'
>>> s.upper()
'BERLIN: 18.4 C AT 4 PM'
```

The examples shown here are just a few of the useful string operations defined in Python. Many more exist, but all the text processing tasks in this course can be accomplished with the operations listed here. In fact, nearly all the tasks we encounter can be solved by using a combination of `split` and `join` in addition to indexing and slicing of strings.

Example; read pairs of numbers (x,y) from a file. To summarize some string operations using an example, consider the task of reading files on the following format;

```
(1.3,0)    (-1,2)    (3,-1.5)
(0,1)      (1,0)     (1,1)
(0,-0.01)  (10.5,-1) (2.5,-2.5)
```

We want to read these coordinate pairs, converted the numbers to floats, and store them as a list of tuples. The algorithm is similar to how we have processed files earlier:

1. Read line by line

2. For each line, split the line into words

3. For each word, strip off the paretheses and split the rest with respect to comma to extract the numbers

From these operations, we may observe that the `split` is a useful tool, as usual when processing text files. For stripping parantheses off the coordinate pairs we can for instance use slicing. Translated into code, the example may look like this:

```
lines = open('read_pairs.dat', 'r').readlines()

pairs = []   # list of (n1, n2) pairs of numbers
for line in lines:
    words = line.split()
    for word in words:
        word = word[1:-1]  # strip off parenthesis
        n1, n2 = word.split(',')
        n1 = float(n1);  n2 = float(n2)
        pair = (n1, n2)
        pairs.append(pair)
```

3 Summary of dictionary and string functionality

The following table lists some useful functionality of dictionaries:

Construction	Meaning
`a = {}`	initialize an empty dictionary
`a = {'point': [0,0.1], 'value': 7}`	initialize a dictionary
`a = dict(point=[2,7], value=3)`	initialize a dictionary w/string keys
`a.update(b)`	add/update key-value pairs from `b` in `a`
`a.update(key1=value1, key2=value2)`	add/update key-value pairs in `a`
`a['hide'] = True`	add new key-value pair to `a`
`a['point']`	get value corresponding to key `point`
`for key in a:`	loop over keys in unknown order
`for key in sorted(a):`	loop over keys in alphabetic order
`'value' in a`	`True` if string `value` is a key in `a`
`del a['point']`	delete a key-value pair from `a`
`list(a.keys())`	list of keys
`list(a.values())`	list of values
`len(a)`	number of key-value pairs in `a`
`isinstance(a, dict)`	is `True` if `a` is a dictionary

The following code summarizes most of the string functionality introduced above:

```
s = 'Berlin: 18.4 C at 4 pm'
s[8:17]          # extract substring
s.find(':')      # index where first ':' is found
s.split(':')     # split into substrings
```

```
s.split()        # split wrt whitespace
'Berlin' in s    # test if substring is in s
s.replace('18.4', '20')
s.lower()        # lower case letters only
s.upper()        # upper case letters only
s.split()[4].isdigit()
s.strip()        # remove leading/trailing blanks
', '.join(list_of_words)
```