

# Organising information: unordered structures

## Author(s)

[Silvio Peroni](#) – [silvio.peroni@unibo.it](mailto:silvio.peroni@unibo.it) – <https://orcid.org/0000-0003-0530-4305>

Digital Humanities Advanced Research Centre (DHARC), Department of Classical Philology and Italian Studies, University of Bologna, Bologna, Italy

## Keywords

Dictionary; Jorge Borges; Infinity; Set

## Copyright notice

This work is licensed under a [Creative Commons Attribution 4.0 International License](#). You are free to share (i.e. copy and redistribute the material in any medium or format) and adapt (e.g. remix, transform, and build upon the material) for any purpose, even commercially, under the following terms: attribution, i.e. you must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use. The licensor cannot revoke these freedoms as long as you follow the license terms.

## Abstract

This chapter introduces the main concepts related to some of the most important data structures for creating and handling sets and dictionaries. The historic hero introduced in these notes is Jorge Luis Borges, considered one of the most famous Argentinian writers of the past century. Among his vast work, he wrote several short stories focussed on the exploration of mathematical concepts and limits.

## Historic hero: Jorge Luis Borges

[Jorge Luis Borges](#), shown in [Figure 1](#), was an Argentine short-story writer, poet, and essayist, who produce several works laying between philosophical literature and fantasy genre. In his short novels, he explored several aspects and situations related to dreams, labyrinths, libraries, mirrors, the notion of infinity, and religions, among others.

One of his most known stories is entitled [The Library of Babel](#) [Borges, 1941]. In this short piece, he describes an incredibly big library, made of hexagonal rooms. In four of the walls of each room, there were 20 bookshelves (5 bookshelves for each wall). Each bookshelf contained 35 books. Each book counted precisely 410 pages of 40 lines each, and each line contained precisely 80 characters. From one of the two empty walls, one could access to a hallway with

two small rooms (one where to sleep standing up, the other with a bathroom) and to the stairs to get to higher hexagonal rooms. The whole library contained all the books ever written by using every possible combination of 25 characters: 22 letters, the period, the comma, and the space. Of course, the main parts of the books contained a nonsense sequence of characters, while others (or even limited parts of them) were, indeed, describing situations using an intelligible language. However, even between those, some books negated the statements of the others explicitly. Thus, the narrator suggested that we are in the presence of all the possible written books. Books that, even when they make sense, are useless because they describe any likely fact from all the possible perspectives. Thus, there is no absolute truth available, and all the possibilities are possible.



**Figure 1.** A picture of Jorge Luis Borges at *L'Hôtel* in Paris. Source: [https://commons.wikimedia.org/wiki/File:Jorge\\_Luis\\_Borges\\_Hotel.jpg](https://commons.wikimedia.org/wiki/File:Jorge_Luis_Borges_Hotel.jpg).

The opening sentence is of particular interest: “[the Library] is composed of an indefinite and perhaps infinite number of hexagonal galleries”. In this passage, the narrator suggests that

there is an infinite number of books in the library, contained in an infinite number of rooms. However, is this the case?

Understanding infinity is a matter of personal perception of things. Often we use to refer to an infinite amount of something (time, space, etc.) when we are speaking about a huge mass of stuff. Stuff that is delimited by some physical or social constraints. The Library of Babel falls in these kinds of situation. Even if the narrator says explicitly that the library is infinite, actually it can contain *only*  $2 \cdot 10^{1834097}$  of books [Foulds, 2017]. We can obtain the previous number by considering all the possible combination of all the finite set of characters. Characters that can appear in all the 40 pages in all the books the library contains. And this number exists, even if it is so big that is not manageable by our mind and, thus, we tend to identify it to infinity, while it is not. As a support to this non-infinity, someone has also tried to [implement the Library as a digital space](#).

Of course, [mathematical infinity](#) exists, as an abstract concept – e.g. think about the set of all the prime numbers, which is an [infinite set](#). However, a computer is physically limited in space and has a limited set of resources. Thus, a computer can only approximate the concept of infinity. For instance, we can implement an algorithm that runs forever in a specific programming language. However, if we ask an electronic computer to run it we will see that it may stop anyway due to some external reasons: a blackout, the breaking of some hardware necessary for the correct execution of its processes, etc.

Considering the applications and implementations of computational systems, we must be aware that infinity (e.g. the infinite tape of a Turing Machine, the endless execution of an algorithm, etc.) is an **illusion**. It is just a theoretical tool which allows us to sketch out possible borders for real-world problems.

## A clarification: classes and methods in Python

In programming languages, [classes](#) are extensible templates for creating objects having a specific type. In practice, all the values (e.g. numbers and strings) and other entities (e.g. lists and stacks) we create are objects of a specific class. The creation of objects of a specified kind is possible by calling a constructor, i.e. a particular function (e.g. `list()` for lists) which creates a new object of that class.

The advantage of [organising all these types of values as classes](#) is that each object made available a set of [methods](#) that allow one to interact with the object itself. A method is a particular kind of function that can be run only if directly called via an object. Their fingertips is structured as follows: `<object>.<method>(<param_1>, <param_2>, ...)`. For instance, methods of the class *list* define all the operations we have introduced for manipulating lists, e.g. `<list>.append(<item>)`, `<list>.remove(<item>)`, etc.

It is possible to create our classes and methods. However, this topic goes beyond the actual scope of this book. If interested in understanding how to create these items, please refer to the documentation provided in the [chapter "Programming languages"](#).

## Unordered structures

In this chapter, we will introduce two specific data structures, that are discussed in detail in the following sections, i.e. *sets* and *dictionaries*. They are among the most basic and used data structures in algorithms (and, more concretely, in programs). They do not specify any order for their elements. Finally, they do not allow repetitions – i.e. the same value cannot be specified twice.

### Sets

A [\*set\*](#) is a countable collection of unordered and non-repeatable elements. It is *countable* because we can use the built-in function `len()` (introduced some chapters ago, when we talked about lists) for counting the elements it contains. Its elements are unordered because the order of the insertion operations does not provide any cardinality relation among such elements. Finally, its elements are not repeatable because the same value cannot be included twice in the set.

Of course, there exist several real examples of such abstract sets in real-life objects. For instance, in [Figure 2](#), we show a class of students and a collection of colours. Both of them are concrete objects that are built starting from the abstract notion of a set.



**Figure 2.** Two examples of a set in real objects: a class of students (left), and a collection of colours contained in a plastic glass (right). Left picture by Uri Tours, source: <https://www.flickr.com/photos/northkoreatravel/10682515504/>. Right picture by Mikel Seijas Alonso, source: <https://www.flickr.com/photos/xumet/2670267503/>.

In Python, a new set can be instantiated using the constructor `set()`. For instance, `my_first_set = set()` will create an empty set and associates it to the variable `my_first_set`.

We can execute several operations on sets, in particular:

- we use the method `<set>.add(<element>)` for adding a new element to the set – for instance, `my_first_set.add(34)` and `my_first_set.add(15)` will add the numbers 34 and 15 to the set – it is worth mentioning that adding an element already included in the set does not add it again;
- we use the method `<set>.remove(<element>)` for removing an element from the set – for instance, `my_first_set.remove(34)` will remove the number 34, obtaining a set with just the element 15 included in it;
- we use the method `<set>.update(<another_set>)` for adding all the elements included in `<another_set>` to the current set – for instance, if we have the set `my_second_set` containing the numbers 1 and 15, `my_first_set.update(my_second_set)` will add just 1 to the current set, since 15 was already present.

```
my_first_set = set() # this creates a new set

my_first_set.add(34) # these two lines add two numbers
my_first_set.add(15) # to the set without any particular order
# currently my_first_set contains two elements:
# set({ 34, 15 })

my_first_set.add("Silvio") # a set can contains element of any kind
# now my_first_set contains:
# set({34, 15, "Silvio"})

my_first_set.remove(34) # it removes the number 34
# my_first_set became:
# set({15, "Silvio"})

# it doesn't add the new elements since they are already included
my_first_set.update(my_first_set)
# current status of my_first_set:
# set({15, "Silvio"})

my_first_set_len = len(my_first_set) # it stores 2 in
my_first_set_len
```

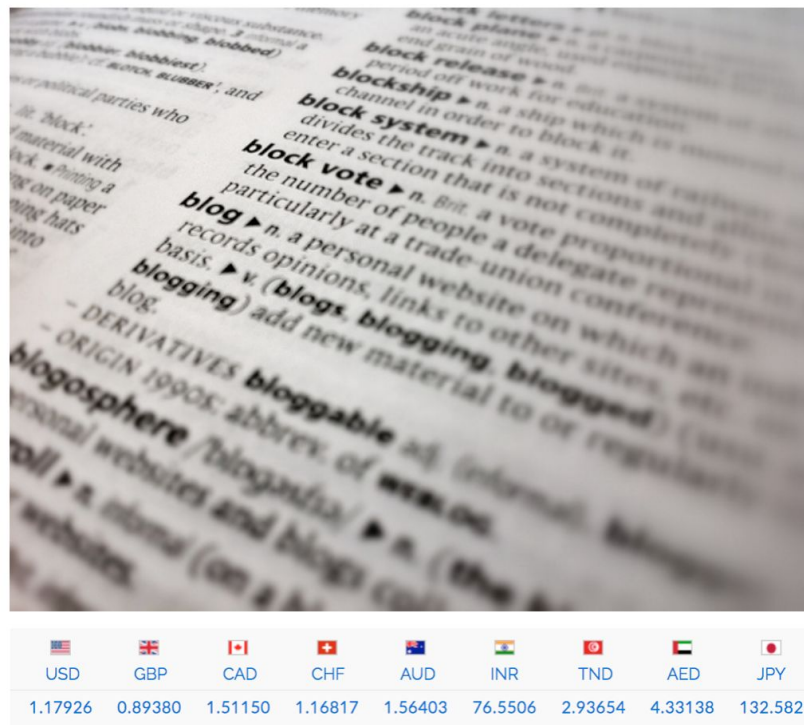
**Listing 1.** How Python allows us to create and handle sets – with numbers and strings. The source code of this listing is available [as part of the material of the course](#).










In [Listing 1](#), we show some examples of the use of sets in Python. As in the examples of the previous chapter, we describe with natural language comments the various aspects related to the creation and modification of sets.

## Dictionaries

A [dictionary](#) is a countable collection of unordered key-value pairs, where the key is non-repeatable in the dictionary. It is countable because we can use the built-in function `len()` for counting the elements it contains. Its elements are unordered because the order of the insertion operations does not provide any cardinality relation among such elements, similar to sets. Finally, the keys of its pairs are not repeatable because the same key cannot be used twice in the dictionary.

Of course, there exist several real examples of such abstract dictionaries in real-life objects. For instance, in [Figure 3](#), we show a collection of definitions and a currency exchange table. Both of them are concrete objects that are built starting from the abstract notion of a dictionary.



								
USD	GBP	CAD	CHF	AUD	INR	TND	AED	JPY
1.17926	0.89380	1.51150	1.16817	1.56403	76.5506	2.93654	4.33138	132.582

**Figure 3.** Two examples of a dictionary in real objects: a collection of definitions (top), and a conversion table from 1 euro to the amount in the other nine different currencies (bottom). Top picture by Doug Belshaw, source: <https://www.flickr.com/photos/dougbelshaw/6877298592/>. Bottom screenshot from <http://www.xe.com/it/>.



```

my_first_dict = dict() # this creates a new dictionary

# these following two lines add two pairs to the dictionary
my_first_dict["age"] = 34
my_first_dict["day of birth"] = 15
# currently my_first_dict contains two elements:
# dict({ "age": 34, "day of birth": 15 })

# a dictionary can contains even key-value pairs of different types
my_first_dict["name"] = "Silvio"
# now my_first_dict contains:
# dict({ "age": 34, "day of birth": 15, "name": "Silvio" })

del my_first_dict["age"] # it removes the pair with key "age"
# my_first_dict became:
# dict({ "day of birth": 15, "name": "Silvio" })

my_first_dict.get("age") # get the value associated to "age"
# the returned result will be None in this case

# the following lines create a new dictionary with two pairs
my_second_dict = dict()
my_second_dict["month of birth"] = 12
my_second_dict["day of birth"] = 28

# it adds a new pair to the current dictionary, and rewrite the value
# associated to the key "day of birth" with the one specified
my_first_dict.update(my_second_dict)
# current status of my_first_dict:
# dict({ "day of birth": 28, "name": "Silvio", "month of birth": 12 })

# it stores 3 in my_first_dict_len
my_first_dict_len = len(my_first_dict)

```

**Listing 2.** How Python allows us to create and handle dictionaries – with numbers and strings as keys and values of pairs. The source code of this listing is available [as part of the material of the course](#).

In Python, a new dictionary can be instantiated using the constructor `dict()`. For instance, `my_first_dict = dict()` will create an empty dictionary and associates it to the variable `my_first_dict`.

We can execute several operations on dictionaries, in particular:

- we use the operation `<dictionary>[<key>] = <value>` for adding a new pair to the dictionary – for instance, `my_first_dictionary["age"] = 34` and `my_first_dictionary["day of birth"] = 15` will add the pairs "age": 34 and "day of birth": 15 to the dictionary – it is worth mentioning that (a) keys must be either strings, numbers, or tuples, and (b) adding a pair with a key that is already included in the dictionary will overwrite the previously-defined pair;
- we use the operation `del <dictionary>[<key>]` for removing the pair identified by the specified key from the dictionary – for instance, `del my_first_dictionary["age"]` will remove the pair "age": 34, obtaining, thus, a dictionary with just the pair "day of birth": 15;
- we use the method `<dictionary>.get(<key>)` for getting the value associated to the pair having the specified key in the dictionary – for instance, `my_first_dictionary.get("day of birth")` will return the number 15, while `my_first_dictionary.get("name")` will return *None* since no pairs in the dictionary include the specified key;
- we use the method `<dictionary>.update(<another_dictionary>)` for adding all the pairs included in `<another_dictionary>` to the current dictionary – for instance, if we have the dictionary `my_second_dictionary` containing the pairs "month of birth": 12 and "day of birth": 28, `my_first_dictionary.update(my_second_dictionary)` will add the pairs "month of birth": 12 to the current dictionary, while the pair "day of birth": 15 will be rewritten with "day of birth": 28.

In [Listing 2](#), we show some examples of the use of dictionaries in Python. In particular, we introduce the effect of using all the operations mentioned above.

## Exercises

1. Write a code in Python to create a set of the following elements: "Bilbo", "Frodo", "Sam", "Pippin", "Merry".
2. Consider the set created in the first exercise, stored in the variable `my_set`. Describe the status of `my_set` after the execution of each of the following operations:  
`my_set.remove("Bilbo"),` `my_set.add("Galadriel"),`  
`my_set.update(set({"Saruman", "Frodo", "Gandalf"})).`
3. Suppose to organise all the elements in the set returned by the second exercise in two different sets: `set_hobbit` that refers to the set `set({"Frodo", "Sam", "Pippin", "Merry"})`, and `set_magician` defined as `set({"Saruman", "Gandalf"})`. Create a dictionary containing two pairs: one that associates the set of hobbits with the key "hobbit", and the other that associates the set of magicians with the key "magician".



# Acknowledgements

The author wants to thank some of the students of the [Digital Humanities and Digital Knowledge second-cycle degree of the University of Bologna](#) – i.e. [Delfina Sol Martinez Pandiani](#), [Eleonora Peruch](#), and [Mattia Spadoni](#) – for having suggested corrections and improvements to the text of this chapter.

# References

Borges, J. L. (1941). La biblioteca de Babel. In El Jardín de senderos que se bifurcan. Editorial Sur. English translation available at <https://sites.evergreen.edu/politicalshakespeares/wp-content/uploads/sites/226/2015/12/Borges-The-Library-of-Babel.pdf>

Foulds, J. (2017). Answer to the question "Is the Library of Babel infinite?". Quora. <https://www.quora.com/Is-the-Library-of-Babel-infinite> (last visited 16 November 2017)