Organising information: unordered structures

Author(s)
Silvio Peroni – silvio.peroni@unibo.it
Department of Classical Philology and Italian Studies, University of Bologna, Bologna, Italy

Keywords
Dictionary; Jorge Borges; Infinity; Set

Abstract

These lecture notes introduce 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 important Argentinian writers of the past century. Among his huge 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 the 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, which was made of hexagonal rooms and where, in four of the walls of each room, there were 20 bookshelves (5 bookshelves for each wall). Each bookshelf contained 35 books. Finally, each book counted exactly 410 pages, each page was organised in 40 lines, and each line contained exactly 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 idea is that the whole library contained all the books that have been and will be written by using every possible combination of 25 basic characters: 22 letters, the period, the comma, and the space. Of course, the main parts of the books contain a nonsense sequence of characters, while others (or even limited parts of them) are, indeed, describing situations using an intelligible language. However, even between those, there were also books that negated explicitly the statements of the others. Thus, the narrator suggested that we are in the presence of all the possible written books that, even when they make sense, are totally useless because they describe any possible fact from all the possible perspectives, and thus there is no absolute truth available and all the possibilities are admissible.

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.

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,

since its very well-explained composition, the library is also made of an infinite number of books, contained in an infinite number of rooms. However, is really this the case?

Understanding infinity is a matter of a personal perception of things. Often we use to refer to an infinite amount of something (time, space, etc.) when actually we are solely speaking about a quite extensive and huge mass of stuff, which still is delimited by some physical or social constraint. 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]. The previous number has been obtained by considering all the possible combination of all the finite set of characters in all the 40 pages in all the books that can be generated. 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.

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, when one tries to make calculations by means of a computer, which is physically limited in space somehow and, thus, in its resources, we can only reach a plausible approximation of it. For instance, even if it is possible to implement an algorithm that runs forever in a specific programming language, technically speaking its implementation runned by a (electronic) computer can stop (and will stop, after all) due to some external reasons – a blackout, the breaking of some hardware necessary for the correct execution of its processes, etc.

When we consider pragmatic applications and implementations of existing 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 certain type. In practice all the values (e.g. numbers and strings) and other entities (e.g. lists and stacks) we create are actually objects of a certain class. The creation of objects of a certain kind is performed by calling a constructor, i.e. a special 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 fingertip is structured as follows: `<object>.<method>(<param_1>, <param_2>, ...)`. For instance, all the operations we have introduced for manipulating lists are actually defined as methods of the class *list*, e.g. `<list>.append(<item>)`, `<list>.remove(<item>)`, etc.

Even if it is possible to create our own classes and methods, this topic goes beyond the actual scope of this course. However, it is possible to understand how to create these items by reading the documentation that has been provided as links in the lecture notes "Programming languages".

Unordered structures

In this lectures, we will introduce two specific data structures, that are discussed in details in the following sections, i.e. sets and dictionaries. Their main characteristics, in addition of being among the most basic and used data structures in algorithms (and, more concretely, in programs), is that they do not specify any order for their elements and that 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 lectures ago, when we talked about lists) for counting the elements it contains. Its elements are unordered because the order of the insertion operations do 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 by means of the constructor `set()`. For instance, `my_first_set = set()` will create an empty set and associates it to the variable `my_first_set`.

Several operations can be done on sets, in particular:

- the method `<set>.add(<element>)` is used 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 that is already included in the set does not add it again;
- the method `<set>.remove(<element>)` is used for removing the 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;
- the method `<set>.update(<another_set>)` is used 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.set("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 lectures, 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.

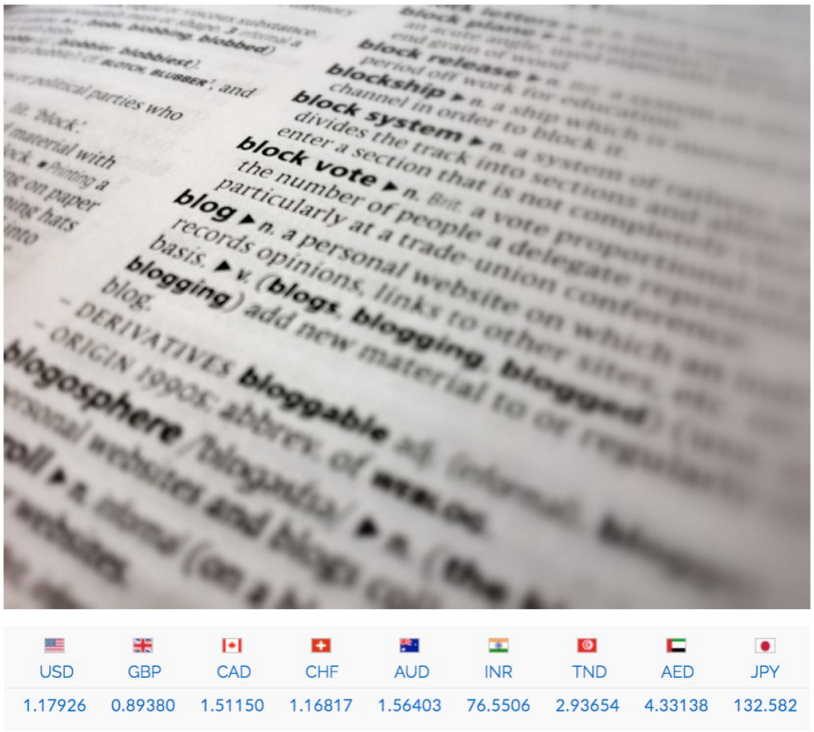

Figure 3. Two examples of a dictionary in real objects: a collection of definitions (top), and a conversion table from 1 euro to the amount in other nine different currencies (bottom). Top picture by Doug Belshaw, source: https://www.flickr.com/photos/dougbelshaw/6877298592/. Bottom screenshot from http://www.xe.com/it/.

```
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_first_dict = dict()
my_first_dict["month of birth"] = 12
my_first_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_dictionary)
# current status of my_first_dict:
# dict({"day of birth": 28, "name": "Silvio", "day of month": 12})

# it stores 3 in my_first_dict_len
my_first_dict_len = len(my_first_dictionary)
```

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 by means of the constructor `dict()`. For instance, `my_first_dict = dict()` will create an empty dictionary and associates it to the variable `my_first_dict`.

Several operations can be done on dictionaries, in particular:

- the operation `<dictionary>[<key>] = <value>` is used 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;
- the operation del `<dictionary>[<key>]` is used 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`;
- the method `<dictionary>.get(<key>)` is used 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 the specified key is not included in any pair of the dictionary;
- the method `<dictionary>.update(<another_dictionary>)` is used 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 aforementioned operations.

Exercises

1. Write a pseudocode in Python so as 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 that all the elements in the set returned by the second exercise have been organised in two different sets, i.e. `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 associate the set of hobbits with the key `"hobbit"`, and the other that associates the set of magicians with the key `"magician"`.

References

Borges, J. L. (1941). La biblioteca de Babel. In El Jardín de senderos que se bifurcan. Editorial Sur. English translation available at http://www.arts.ucsb.edu/faculty/reese/classes/artistsbooks/The%20Library%20of%20Babel.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)