Organising information: ordered structures

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

Keywords
Donald Knuth; List; Queue; Stack

Abstract

These lecture notes introduce the notion of ordered data structures, i.e. some basic containers of elements that can be used to organise data in a specific way. The historic hero introduced in these notes is Donald Knuth, who has been one of the most relevant scientists and contributors on the topic of the formal analysis of the computational complexity of algorithms.

Historic hero: Donald Knuth

Donald Ervin Knuth (shown in Figure 1) is one of the most important Computer Scientist of the past 50 years. He is one of the main contributors to the theoretical and practical development of the analysis of the computational complexity of algorithms, that we have introduced in the previous lectures. Among his huge list of contributions, there are especially two works that deserve a particular attention in this context: its series of monographs about algorithms and their analysis entitled *The Art of Computer Programming*, and the TeX typesetting system for writing academic documents, that has been used to write the aforementioned series and it is one of the most used tools for communicating and publishing scientific results in academia.

According to several experts, the series of monographs he has written is one of the most comprehensive works to programming and algorithmics. The project started in 1962 as a twelve-chapter book, and then split in a series of seven volumes, of which only the first four

have been published so far – while the others are still in writing. The first volume is entirely dedicated to the mathematical foundations for allowing a formal analysis of algorithms, and to a comprehensive introduction of all the basic data structures.

Figure 1. Donal Knuth in 2005. Picture by Jacob Appelbaum, source: https://commons.wikimedia.org/wiki/File:KnuthAtOpenContentAlliance.jpg.

Data structures are the possible ways in which we organise the information to be processed and returned by a computer, so as it can be accessed and modified in an efficient and computational manner. In practice, a data structure is a sort of bucket where we can place some information, that provides some methods to add and retrieve pieces of such information. The most simple data structures are lists and, as such, they are the first data structures introduced in Knuth's first volume of *The Art of Computer Programming* [Knuth, 1997]. They are, probably, one of the most important building blocks of algorithms, since they have plenty of applications.

Functions in Python

In the previous lecture notes, we showed to use `def <algorithm>(<param_1>, <param_2>, ...)` for implementing algorithms. As anticipated, we actually provided a mechanism for implementing *functions* in Python. Functions are a common feature of any programming language, since they provide a mechanism for listing a sequence of instructions (which implements an algorithm) under a particular name, so as to organise a block of reusable code to solve a particular computational problem.

In Python, like in other programming languages, we can split functions into two different sets: *built-in* functions, and *user-defined* functions. Built-in functions are the ones that are made available by the programming language itself, and that can be reused for addressing a particular task on some values. For instance, the function `len()`, that can be used to count the items in a collection (e.g. how many characters are included in a string), or the constructor `list()`, that we will introduce in this lecture notes, are functions of this type. The other kind of functions, i.e. the user-defined ones (e.g. your algorithm), groups all the functions written by a user of the language for addressing some specific requirements or tasks that are not addressable by means of one built-in function directly. All the algorithms we have introduced in the past lecture notes comply with this latter kind of functions. In fact, they can be seen as user-defined functions.

All the functions, either built-in or user-defined, can be run. Some of those may be run without specifying any input values – e.g. the aforementioned constructor for lists – and return a new object of a specific kind. Others, instead, need to be run by specifying the necessary input values, such as `len(<string>)`. One of the most used and important functions of this kind is `print(<object_1>, <object_2>, ...)`. This function is very useful since it allows one to print on screen a particular value (that can be referred by a variable).

```
def add_one(n):  # define a function
    return n + 1

result = add_one(41)  # run the function specifying 41 as input
print(result)  # print the result stored in the variable 'result'
```

Listing 1. The definition of a simple function and its execution using *41* as the input value. The result of its execution is then stored in a variable and printed on the screen. The source code of this listing is available as part of the material of the course.

The mechanism used in Python for running a function is just to call it by its name and by specifying the required input values, if any – as already shown in the previous lecture notes for testing the algorithms developed. For instance, Listing 1 is showing the definition of a simple

function. The code defined by the function will not run until it is explicitly requested, i.e. when it is called by specifying *41* as input.

Additional functions and variables that can be used in Python are actually loaded when needed by importing the *package* that contains them. Packages are just a mechanism to expose Python modules. We can consider a module like a Python file (extension `.py`) that contains the definition of variables, functions, and even runnable code. They are organised hierarchically in directories, where each directory can be defined as a package.

The basic installation of Python makes available a huge set of packages for addressing several operations and functions. For instance, the constructor for creating stacks and queues, that we will introduce in this lecture notes (i.e. *deque()*) is actually contained in a module of the package *collections*. Thus, for using it in Python, it is necessary to import the module (or a function, or a variable) by means of the following command: from `<package> import <module or function or variable>` – e.g. `from collections import deque`.

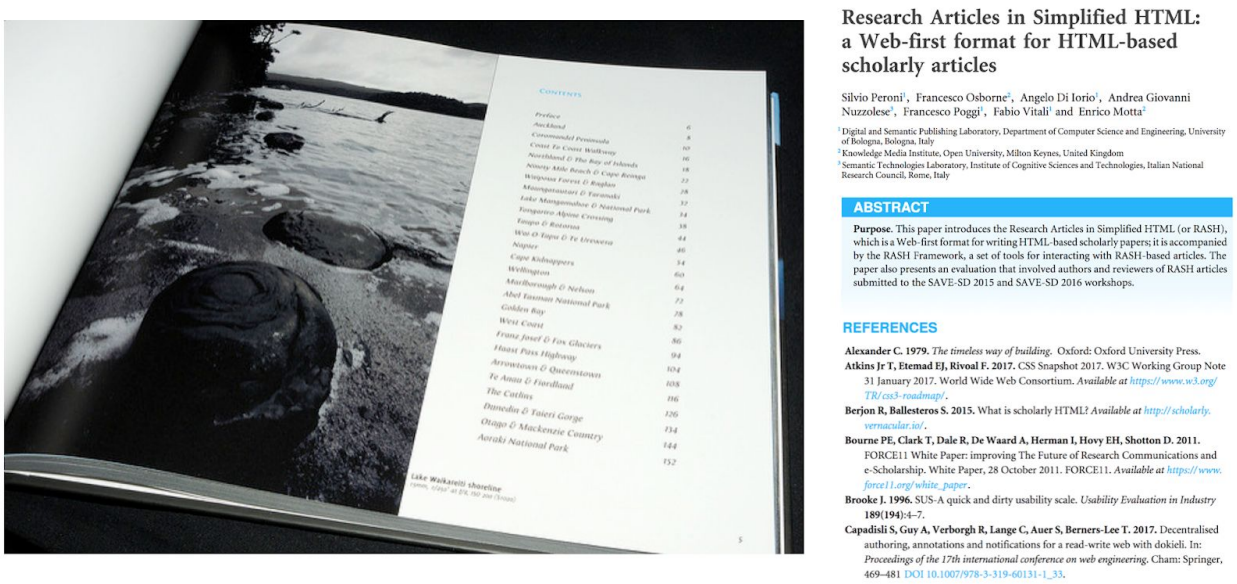
Ordered data structures

In this lectures, we will introduce three specific data structures, that are discussed in details in the following sections, i.e.: lists, stacks, and queues. Their main characteristic, in addition of being among the most basic and used data structures in algorithms (and, more concretely, in programs), is that the order in which their elements have been added matters. In fact, they interlink all their elements so as to create an ordered chain of elements which allows us to have a clear prediction of the behaviour of the addition and removal operations they make available.

Lists

A *list* is a countable sequence of ordered and repeatable elements. It is *countable* because there is a proper way of knowing the length of the list (i.e. how many elements it contains). In particular, Python makes available a function, i.e. `len(<countable_object>)`, that takes a countable element as input (like a list), and returns the number of elements that it contains. Its elements are *ordered* because they are placed in the list in a specific order, which is preserved even if we add or remove particular elements. Its elements are also *repeatable* since they may appear more than one time in the list.

Of course, there exist several real examples of such abstract lists in real-life objects. For instance, in Figure 2, we show a table of content of a book and a bibliographic reference list of an article. Both of them are concrete objects that are built starting from the abstract notion of a list.



Research Articles in Simplified HTML: a Web-first format for HTML-based scholarly articles

Silvio Peroni[1], Francesco Osborne[2], Angelo Di Iorio[1], Andrea Giovanni Nuzzolese[3], Francesco Poggi[1], Fabio Vitali[1] and Enrico Motta[2]

[1] Digital and Semantic Publishing Laboratory, Department of Computer Science and Engineering, University of Bologna, Bologna, Italy
[2] Knowledge Media Institute, Open University, Milton Keynes, United Kingdom
[3] Semantic Technologies Laboratory, Institute of Cognitive Sciences and Technologies, Italian National Research Council, Rome, Italy

ABSTRACT

Purpose. This paper introduces the Research Articles in Simplified HTML (or RASH), which is a Web-first format for writing HTML-based scholarly papers; it is accompanied by the RASH Framework, a set of tools for interacting with RASH-based articles. The paper also presents an evaluation that involved authors and reviewers of RASH articles submitted to the SAVE-SD 2015 and SAVE-SD 2016 workshops.

REFERENCES

Alexander C. 1979. *The timeless way of building.* Oxford: Oxford University Press.

Atkins Jr T, Etemad EJ, Rivoal F. 2017. CSS Snapshot 2017. W3C Working Group Note 31 January 2017. World Wide Web Consortium. *Available at https://www.w3.org/TR/css3-roadmap/.*

Berjon R, Ballesteros S. 2015. What is scholarly HTML? *Available at http://scholarly.vernacular.io/.*

Bourne PE, Clark T, Dale R, De Waard A, Herman I, Hovy EH, Shotton D. 2011. FORCE11 White Paper: improving The Future of Research Communications and e-Scholarship. White Paper, 28 October 2011. FORCE11. *Available at https://www.force11.org/white_paper.*

Brooke J. 1996. SUS-A quick and dirty usability scale. *Usability Evaluation in Industry* 189(194):4–7.

Capadisli S, Guy A, Verborgh R, Lange C, Auer S, Berners-Lee T. 2017. Decentralised authoring, annotations and notifications for a read-write web with dokieli. In: *Proceedings of the 17th international conference on web engineering.* Cham: Springer, 469–481 DOI 10.1007/978-3-319-60131-1_33.

Figure 2. Two examples of a list in real objects: the table of contents of a book (left), and a bibliographic reference list in a research paper (right). Left picture by Marcus Holland-Moritz, source: https://www.flickr.com/photos/mhx/4347706564/. Right screenshot, source: https://doi.org/10.7717/peerj-cs.132.

In Python, a new list can be instantiated by means of the constructor `list()`. For instance, `my_first_list = list()` will create an empty list and associates it to the variable `my_first_list`. A list can be seen as a left-to-right sequence of elements, where the left-most element identifies the head of the list, while the last one represents the tail of the list. Several operations can be done on lists, in particular:

- the method `<list>.append(<element>)` is used for adding a new element to the list – for instance, `my_first_list.append(34)` and `my_first_list.append(15)` will add the number *34* to the list, and the number *15* as follower of the previous one;
- the method `<list>.remove(<element>)` is used for removing the first instance of an element in the list – for instance, `my_first_list.remove(34)` will remove the first number *34* which is encountered by scanning the list from its begin (i.e. from the first-added elements to the last-added ones), obtaining, thus, a list with just the element *15* included in it;
- the method `<list>.extend(<another_list>)` is used for adding all the elements included in `<another_list>` to the current list – for instance, if we have the list `my_second_list` containing the numbers *1* and *83*, `my_first_list.extend(my_second_list)` will add *1* and *83* as followers of *15*.

In Listing 2, we show some examples of the use of lists in Python. In these examples, we describe with natural language comments (introduced by a `#`) the various aspects related to the creation and modification of lists.

```
my_first_list = list()  # this creates a new list

my_first_list.append(34)  # these two lines add two numbers
my_first_list.append(15)  # to the list in this precise order
# currently my_first_list contains two elements:
# list([ 34, 15 ])

# a list can contains element of any kind
my_first_list.append("Silvio")
# now my_first_list contains:
# list([34, 15, "Silvio"])

# it removes the first instance of the number 34
my_first_list.remove(34)
# my_first_list became:
# list([15, "Silvio"])

# it add again all the elements in my_first_list to the list itself
my_first_list.extend(my_first_list)
# current status of my_first_list:
# list([15, "Silvio", 15, "Silvio"])

# it stores 4 in my_first_list_len
my_first_list_len = len(my_first_list)
```

Listing 2. How Python allows us to create and handle lists – with numbers and strings. The source code of this listing is available as part of the material of the course.

Stacks

A stack is a kind of list seen from a particular perspective, i.e. from bottom to top, and with a specific set of operations. Figure 3 shows two different example of stacks in real-life objects. We have a stack of chairs (left) and a pile of books (right).

The main characteristic of the elements of this structure is that they follow a *last in first out* strategy (*LIFO*) for addition and removal. Basically, it means that the last element inserted in the structure is placed in the top of the stack and, thus, it is also the first one that will be removed when requested. In addition, to obtain the element placed, for instance, in the middle of the stack, we necessarily need to remove all the elements that have been added *after* such middle element, from the most recent elements to the eldest ones.

In Python, a new stack can be instantiated by means of the constructor `deque()` – included in the `collections` module, to be imported. For instance, `my_first_stack = deque()` will create an empty stack and associates it to the variable `my_first_stack`.

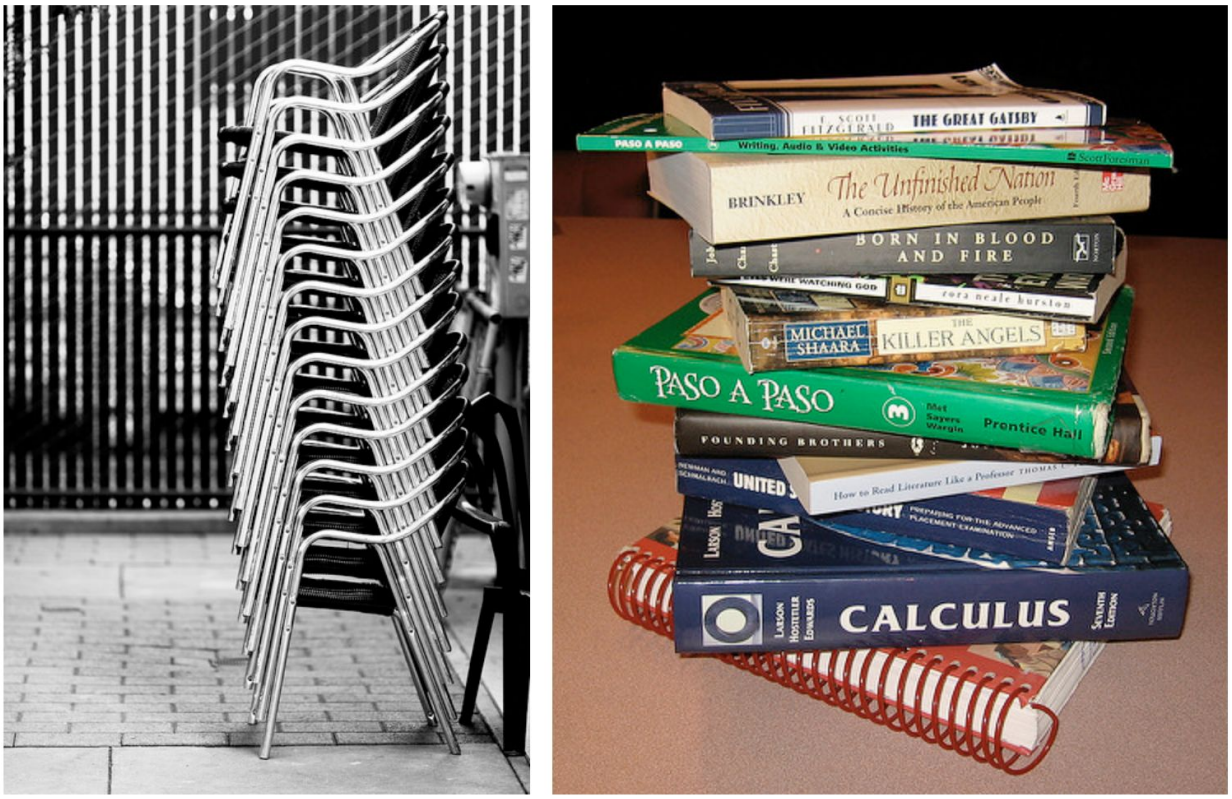


Figure 3. Two examples of a stack of real objects: a stack of chairs (left), and a pile of books (right). Left picture by Jeremy Brooks, source: https://www.flickr.com/photos/jeremybrooks/16410797960/. Right picture by Cary Lee, source: https://www.flickr.com/photos/the1andonlycary/3310345438/.

Three main operations can be done on stacks, in particular:

- the method `<stack>.append(<element>)` is used for adding a new element on the top of the stack – for instance, `my_first_stack.append(34)` and `my_first_stack.append(15)` will add the number *34* to the stack, and the number *15* upon previous one;
- the method `<stack>.pop()` is used for removing the element on the top of the stack, that will be returned – for instance, `my_first_stack.pop()` will remove the number *15* and will be returned as well, obtaining, thus, a stack with just the element *34* included in it;
- the method `<stack>.extend(<another_stack>)` is used for adding all the elements included in `<another_stack>` on the top of the current stack – for instance, if we have the stack `my_second_stack` containing the numbers *1* and *83*, `my_first_stack.extend(my_second_stack)` will add *1* and *83* on top of *34*.

In Listing 3, we show some examples of the use of stacks in Python. In particular, we organise some books (actually, their titles) written by Neil Gaiman in a stack.

```
from collections import deque  # import statement

my_first_stack = deque()  # this creates a new stack

my_first_stack.append("Good Omens")  # these lines add two books
my_first_stack.append("Neverwhere")
# currently my_first_stack contains two elements:
#        "Neverwhere"])
# deque(["Good Omens",

my_first_stack.append("The Name of the Rose")  # add a new book
# now my_first_stack contains:
#        "The Name of the Rose")]
#        "Neverwhere",
# deque(["Good Omens",

my_first_stack.pop()  # it removes the element on top of the stack
# my_first_stack became:
#        "Neverwhere"])
# deque(["Good Omens",

my_second_stack = deque()  # this creates a new stack
my_second_stack.append("American Gods")  # these lines add two books
my_second_stack.append("Fragile Things")
# currently my_second_stack contains two elements:
#        "Fragile Things"])
# deque(["American Gods",

# it add all the elements in my_second_stack on top of my_first_stack
my_first_stack.extend(my_second_stack)
# current status of my_first_stack:
#        "Fragile Things"])
#        "American Gods",
#        "Neverwhere",
# deque(["Good Omens",
```

Listing 3. How Python allows us to create and handle stacks – with book titles. The source code of this listing is available as part of the material of the course.

Queue

A queue is a kind of list seen by another perspective, i.e. from left to right, and with a specific set of operations. Figure 4 shows two different example of queues in real-life objects. We have a queue of children (left) and a line of cabs (right).

The main characteristic of the elements of this structure is that they follow a *first in first out* strategy (*FIFO*) for addition and removal. Basically, it means that the first element inserted in the structure is placed in the left-most part of the queue and, thus, it is also the first one that will be removed when requested. Similar to stacks, even in queues it is necessary to remove all the elements that have been added *before* a certain target element – i.e. from the eldest elements to the most recent ones – to obtain it.

Figure 4. Two examples of a queue of real objects: a queue of children waiting their turn for playing with a slide (left), and a cab wait line (right). Left picture by Prateek Rungta, source: https://www.flickr.com/photos/rungta/4409560365/. Right picture by Lynda Bullock, source: https://www.flickr.com/photos/just1snap/5141019486/.

In Python, a new queue can be instantiated by means of the constructor `deque()`, which is the same used for stacks. Thus, it is the way one uses it that classifies the object instantiated as a stack or a queue. Thus, as before, `my_first_queue = deque()` will create an empty queue and associates it to the variable `my_first_queue`. Three main operations can be done on queues, in particular:

- the method `<queue>.append(<element>)` is used for adding a new element at the first available position in the queue, i.e. from the right of the queue – for instance,

`my_first_queue.append(34)` and `my_first_queue.append(15)` will add the number *34* to the queue as the first element, and the number *15* after the previous one;

- the method `<queue>.leftpop()` is used for removing the first element of the queue, i.e. the first appended, that will be returned – for instance, `my_first_queue.leftpop()` will remove the number *34* that will be returned, obtaining, thus, a queue with just the element *15* included in it;
- the method `<queue>.extend(<another_queue>)` is used for adding all the elements included in `<another_queue>` after (i.e. on the right of) those ones already included in the current queue – for instance, if we have the queue `my_second_queue` containing the numbers *1* and *83*, `my_first_queue.extend(my_second_queue)` will add *1* and *83* after *34*.

```
from collections import deque  # import statement

my_first_queue = deque()  # this creates a new queue

my_first_queue.append("Vanessa Ives")  # these lines add two people
my_first_queue.append("Mike Wheeler")
# currently my_first_queue contains two elements:
# deque(["Vanessa Ives", "Mike Wheeler")

my_first_queue.append("Eleven")  # add a new person
# now my_first_queue contains:
# deque(["Vanessa Ives", "Mike Wheeler", "Eleven"])

my_first_queue.popleft()  # it removes the first element added
# my_first_queue became:
# deque(["Mike Wheeler", "Eleven"])

my_second_queue = deque()  # this creates a new queue
my_second_queue.append("Michael Walsh")     # these lines add two
people
my_second_queue.append("Lawrence Cohen")
# currently my_second_queue contains two elements:
# deque(["Michael Walsh", "Lawrence Cohen"])

# add all the elements in my_second_queue at the end of
my_first_queue
my_first_queue.extend(my_second_queue)
# current status of my_first_queue:
#  deque(["Mike Wheeler", "Eleven", "Michael Walsh", "Lawrence
Cohen"])
```

Listing 4. How Python allows us to create and handle queues – with people. The source code of this listing is available as part of the material of the course.

In Listing 4, we show some examples of the use of queues in Python. In particular, we organise in a queue some (fictional) people that are waiting in the library so as to borrow some books.

Exercises

1. Write a sequence of instructions in Python so as to create a list with the following elements ordered alphabetically: `"Harry"`, `"Draco"`, `"Hermione"`, `"Ron"`, `"Severus"`.
2. Consider to have a stack obtained by processing, one by one, the elements included in the list of the first exercise, i.e. `my_stack = deque(["Draco", "Harry", "Hermione", "Ron", "Severus"])`. Describe the status of my_stack after the execution of each of the following operations: `my_stack.pop()`, `my_stack.pop()`, `my_stack.append("Voldemort")`.
3. Consider to have a queue obtained by processing, one by one, the elements included in the list of the first exercise, i.e. `my_queue = deque(["Draco", "Harry", "Hermione", "Ron", "Severus"])`. Describe the status of my_queue after the execution of each of the following operations: `my_queue.popleft()`, `my_queue.append("Voldemort")`, `my_queue.popleft()`.

Acknowledgements

I would like to thank one of the students of the course, Severin Josef Burg, for having suggested corrections to the text of these lecture notes.

References

Knuth, D. (1997). The Art of Computer Programming, Vol. 1: Fundamental Algorithms. 3rd Edition. Addison-Wesley Professional. ISBN: 978-0201896831. Also available at http://broiler.astrometry.net/~kilian/The_Art_of_Computer_Programming%20-%20Vol%201.pdf