Organising information: ordered structures

Author(s)
Silvio Peroni – 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
Donald Knuth; List; Queue; Stack

Abstract

This chapter introduces the notion of ordered data structures, i.e. some primary 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 [Roberts, 2018]. 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 previous chapters. His contributions include the series of monographs about algorithms and their analysis entitled *The Art of Computer Programming* and the [TeX](TeX) typesetting system for writing academic documents. TeX has been used to write the series mentioned above, 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. It was 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 fundamental data structures.

Figure 1. Donald 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 such information can be accessed and modified efficiently and computationally. 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 crucial building blocks of algorithms since they have plenty of applications.

Functions in Python

In the previous chapter, we showed to use `def <func_name>(<parameter_1>, <parameter_2>, ...)` for implementing algorithms. As anticipated, we provided a mechanism for implementing *functions* in Python. Functions are a common feature of any programming language. They provide a mechanism for listing a sequence of instructions (which implements an algorithm) under a particular name, 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. We can use them for addressing a particular task on some values. For instance, the function `len()` can be used to count the items in a collection (e.g. how many characters a string contains). The constructor `list()` that we will introduce in this chapter is used to create a new list object. All such functions are built-in functions.

The other kind of functions are those defined by a user (e.g. the algorithms in the previous chapters). They refer to all the functions written by a user of the language for addressing some specific requirements or tasks that are not addressable using one built-in function directly. All the algorithms we have introduced in the previous chapter comply with this latter kind of functions. We, as users of Python, have defined them.

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 constructor mentioned above 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 handy since it allows one to [print](#) on screen a particular value (even when 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 to call it using its name and by specifying the required input values (if any). In the previous chapters, we showed such calling for testing the algorithms developed. For instance, [Listing 1](#) shows the definition of a simple

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

In Python, we can use additional functions and variables that are 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 includes the definition of variables, functions, and even runnable code. They are organised hierarchically in directories, where each directory defines a package.

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

Ordered data structures

In this chapter, we will introduce three specific data structures, discussed in detail in the following sections: lists, stacks, and queues. They are among the most basic and used data structures in algorithms (and, more concretely, in programs). Their main characteristic is that the order in which their elements have been added matters. They organise all their items in an ordered chain which allows us to have a precise 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 items 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 items that it contains. All the items in the list follow a precise order that does not change 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 contents 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.

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, we can instantiate a new list through the constructor `list()`. For instance, `my_first_list = list()` will create an empty list and associates it to the variable `my_first_list`. We can describe a list 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. We can execute several operations on lists, in particular:

- the method `<list>.append(<item>)` allows one to add a new item 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(<item>)` allows one to remove the first instance of an item 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 items to the last-added ones), obtaining, thus, a list with just the element *15* included in it;
- the method `<list>.extend(<another_list>)` allows one to add all the items 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 items:
# list([ 34, 15 ])

# a list can contains item 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 items 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 examples of stacks in real-life objects. We have a stack of chairs (left) and a pile of books (right).

The main characteristic of the items of this structure is that they follow a *last in first out* strategy (*LIFO*) for addition and removal. It means that the previous item inserted in the data structure is available in the top of the stack. Thus, it is also the first one that we remove when requested. To obtain the item placed, for instance, in the middle of the stack, we necessarily need to remove all the items added after such middle item, from the most recent items to the eldest ones.

In Python, a new stack can be instantiated using the constructor `deque()` – included in the `collections` module, to import. For instance, `my_first_stack = deque()` will create an empty stack and associates it to the variable `my_first_stack`.
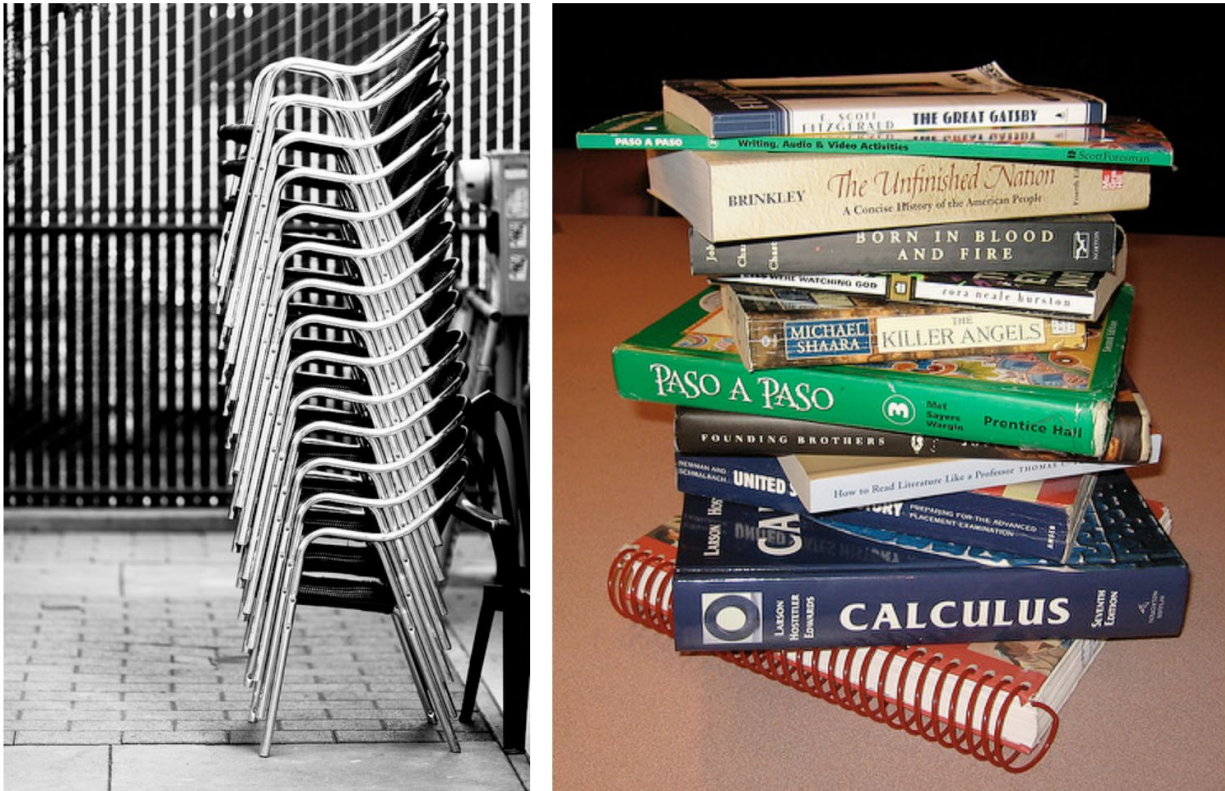
Figure 3. Two examples of a stack of real objects: a stack of chairs (left), and a pile of books (right). Left picture by Jeremy Brooks, source:
https://www.flickr.com/photos/jeremybrooks/16410797960/. Right picture by Cary Lee, source:
https://www.flickr.com/photos/the1andonlycary/3310345438/.

We can execute three main operations on stacks:

- the method `<stack>.append(<item>)` allows one to add a new item 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()` allows one to remove the item 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 item *34* included in it;
- the method `<stack>.extend(<another_stack>)` allows one to add all the items 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 items:
#          "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 item 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 items:
#          "Fragile Things"])
# deque(["American Gods",

# it add all the items 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 examples of queues in real-life objects. We have a queue of children (left) and a line of cabs (right).

The main characteristic of the items of this structure is that they follow a *first in first out* strategy (*FIFO*) for addition and removal. It means that we place the first item inserted in the data structure in the left-most part of the queue. Thus, it is also the first one that we can remove when requested. Similar to stacks, even in queues it is necessary to remove all the items added *before* a specific target item – 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, we can instantiate a new queue through 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`.

We can execute three main operations on queues:

- the method `<queue>.append(<item>)` allows one to add a new item 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 item, and the number *15* after the previous one;
- the method `<queue>.leftpop()` allows one to return the first item 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 item *15* included in it;

- the method `<queue>.extend(<another_queue>)` allows one to add all the items 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")  # add two people
my_first_queue.append("Mike Wheeler")
# currently my_first_queue contains two items:
# 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 item added
# my_first_queue became:
# deque(["Mike Wheeler", "Eleven"])

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

# add the items 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 use the queue to list some (fictional) people waiting their turn in the library to borrow some books.

Exercises

1. Write a sequence of instructions in Python 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

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

Roberts, S. (2018). The Yoda of Silicon Valley. The New York Times. Retrieved from https://www.nytimes.com/2018/12/17/science/donald-knuth-computers-algorithms-programming.html (last visited 26 October 2019)