Chapter 5

# Array-like structures

Elementary data types are the basic way for organizing data. Variables allow us to organize data processing procedures based on the types of data. For a strong typed language, this could identify a large number of design errors during compilation. Even with loosely typed languages, variable typing provides insurance against applying wrong processing to wrong data.

However, data is not always *packaged* in separated containers. We frequently need to divide data into similar pieces before applying algorithms to it. This is where compound data structures are beneficial. Programming languages provide a wide range of complex data structures, yet arrays remain one of the most popular. Arrays are particularly helpful in algorithms that analyze data.

In this chapter, we will explore array-like structures since they are common in programming languages. We will also cover *dictionaries* and *strings* as special cases of arrays. It is worth noting that array-like structures in Python are far more richer than those described in this chapter. So, the word array would not be particularly appropriate to represent the Python structures presented in this chapter, as the most exact word would be *sequences*.

## 1. Arrays

Arrays in the algorithmic language differ significantly from those in Python. The basic type of arrays in Python is `List`, which refers to structures that differ significantly from arrays in the algorithmic language. For both languages, we will use distinct array definitions.

### 1.1. Arrays in the algorithmic language

An array is a collection of elements of the same type stored in a contiguous block of memory. Arrays provide random access, which is an index-based access where an index indicates the rank of data in the array. An array is characterized by the type of its elements and its size, which is the number of elements contained therein. The size of an array in the algorithmic language is fixed, and it is part of its type.

Arrays are declared by this syntax:

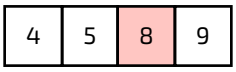
algo

```
1 var arr:array[n] of type
```

In this code, `n` is a constant, and `type` is any type supported by the language (including arrays themselves). To access an element in an array, either in reading or writing, we use the syntax `arr[e]` where `e` is an expression that is evaluated to an integer. For instance, `arr[0]` is the first element, the second element is `arr[1]` and so on. The last element is `arr[n-1]`.

In the algorithmic language, never access an array element without first determining if the index is within the accepted range. If the index is negative or exceeds the array size, this is an error.

Schematically, an array is represented as contiguous block of data. Consider the array `[4,5,8,9]`, it is stored in memory as (the pink box indicates the element at the index 2):




Let's consider a simple example of an algorithm that reads $n$ integers then prints them out in the reverse order.

algo

```
1  algorithm reverse_print
2  const n=15
3  var i:integer
4  var arr:array[n] of integer
5  begin
6      for i=0 to n-1 do
7          arr[i]=integer(input("Enter a value:"))
8      for i=n-1 to 0 step -1 do
9          print(arr[i])
10 end
```

This example shows how to loop across arrays. A basic index-based loop is sufficient to go over all array elements. However, in most situations, nested loops are employed to do more complex processing.

An array should always be initialized before use.

Arrays are structures that can be compared. The algorithmic language has no special syntax for that, the programmer has to write its own code. Let's make a code that decide if two arrays (of the same type) are equals:

algo

```
1  …
2  const n=…
3  var eq:boolean
4  var i:integer
5  var arx,ary:array[n] of integer
6  begin
7      …
8      eq=true
9      i=0
10     while eq and i < n do
11     begin
12         if arx[i]!=ary[i] then eq=false
13         else i=i+1
14     end
15     if eq then print("Arrays are equal")
16     else print("Arrays are no equal")
17 end
```

This code can be shortened a little bit with a `for` loop:

algo

```
1  …
2  const n=…
3  var eq:boolean
4  var i:integer
5  var arx,ary:array[n] of integer
6  begin
7      …
8      eq=true
9      for i=0 to n-1 do
10         if arx[i]!=ary[i] then
11         begin
12             eq=false
13             break
14         end
15     if eq then print(“Arrays are equal”)
16     else print(“Arrays are no equal”)
17 end
```

Let’s take another example in which we compute the number of elements above the average of the values in an array:

algo

```
1  …
2  const n=…
3  var i,nb:integer
4  var arx:array[n] of integer
5  var avg:float begin
6      …
7      for i=0 to n-1 do
8          avg=avg+arx[i]
9      avg=avg/n
10     for i=0 to n-1 do
11         if arx[i]>avg then nb=nb+1
12     print(“Count of elements above the average”,nb)
13 end
```

## 2. Lists in Python

Lists in Python are more powerful than arrays in the algorithmic language. But this has a cost since for some applications lists can a little bit slow. In reality, it is possible to use arrays in Python through the module `numpy`, but this will not be covered in this course. Still, `numpy` arrays are more efficient and more suitable for numerical calculations.

In Python, a list can contain any type of data. It is possible to mix integers, with booleans, with strings, with arrays themselves, and so on. For sake of clarity, we will just use lists containing the same type of elements. Note that another major difference between arrays in the algorithmic language and Python is that lists can created and manipulated in different ways and that size is dynamic. That is, we can change their size by adding or removing elements. Finally, loops can be done in several ways on lists. Comprehension and filtering are one of the most powerful. We will just cover comprehension.

### 2.1. List constructors in Python

List can basically be built using three ways:

- Using the constructor `list(iterable)` which builds a new list for an iterable (an object that can be used in a loop). For instance, we can write `list(range(10))`.
- Using the constructor `[...]`. For instance, we can write `[0,1,2,3,4,5,6,7,8,9]`. In particular, the syntax `[]` builds an empty list.
- Using the multiplication operator `[...]*nb` where `nb` is an integer value. For example, `[0]*10` builds an array the elements of which are all zero. This syntax can be a source of error if the initial list contains arrays.

To access the elements of an array, we use the syntax `l[index]` (as in the algorithmic language). However, negative indices are possible. `l[-1]` is the last element, `l[-2]` is the penultimate element of the list, and so on. The size of a list (or any sequence) is got by the function `len(l)`.

Let's rewrite the previous array algorithms using Python. We begin with the reverse printing program (to declare the type of a list, we have to use `from typing import List`):

```py
...
n:int=15
arl:List[int]=[0]*n
for i in range(n):
    arl[i]=input("Give a new value")
for i in range(n-1,-1,-1):
    print(arl[i],end="")
```

The comparison of two lists is made by:

```py
...
eq=True
for i in range(len(arx)):
    if arx[i]==ary[i]:
        eq=False
        break
if eq:print("Arrays are equal")
else:print("Arrays are no equal")
```

The count of elements above the average in a list is made by:

```py
...
avg=sum(arx)/len(arx)
nb=0
for e in arx:
    if e>avg:nb+=1
print("Count of elements above the average",nb)
```

In this example, we see many new elements:

- The function `sum` is predefined in Python and computes the sum of the elements of a list. We can also the functions `max` and `min`.
- Iterating overs lists. In Python, looping over a list means that the loop variable will take consecutively the values in the list. For instance, if we have `arx=[1,2,3]` then the loop will assign the value 1 to `e`, then the value 2, then the value 3. This form is used if we are just interested with the elements of an array. If we are interested in both indices and elements of an array, then we use the syntax `for i,e in enumerate(i,e)`.

Note that we won't cover dynamic sized lists in this course or list comprehension or filtering, as it ia an introductory course.

## 2.2. Tuples

Tuples are similar to lists, but are immutable. Once a tuple has been built, neither its size nor its elements can be changed; otherwise, a runtime error is raised. Tuples are created with parenthesis rather than brackets. For example, the tuple `(1,2,3)` defines an immutable *list* containing elements 1, 2, and 3. All list operations are also applied to tuples except modification.

A special intention should be paid for a one-element tuple. In fact, the notation `(1)` is equivalent to `1` and does not build a tuple. We should use an extra comma to tell the compiler that we want to build a tuple `(1,)`.

## 2.3. Packing and unpacking

Lists and tuples offer an elegant way to combine data into a single data unit, but also a way to to *scatter* it. These are referred to as the *pack* and *unpack* operations.

Packing is just grouping data into a list or tuple. This can be done by utilizing the `[...]` or `(...)` syntax. Unpacking is the reverse operation of taking a list or tuple and dividing it into single values. We have previously seen how unpacking works for multiple assignments. In the assignment `x,y=a,b`, `a` is assigned to `x` and `b` to `y`. In reality, the left and right sides of this assignment are just tuples, therefore this assignment is equivalent to `(x,y)=(a,b)`.

Remember what we stated about the number of variables on the left and right sides of an assignment; they should be equal. If not, we can use a dummy variable (_) on the left side, but this still counts for one variable. Assume we have a list `l=[1,2,3,4]` and wish to extract the first, second, and rest of the list into three variables: `a`, `b`, and `c`. Because there are three variables but four elements in the list, writing `a,b,c=l` will result in an error. We should use _ twice to prevent the problem, but this is not what we want to do. Python provides a special and powerful syntax for packing elements as lists. The code `a,b,*c=l` assigns 1 to `a`, 2 to `b`, and the remaining list (`[3,4]`) to `c`. We may also write `*a,b,c=l`, which assigns `[1,2]` to `a`, 3 to `b`, and 4 to `c`.

**Remark:** There can be only one starred expression in a statement.

# 3. Strings

String are special cases of arrays consisting only of characters. They are generally immutable in order to make them efficient. String are usually used to represent a text.

## 3.1. Strings in the algorithmic language

A string variable is declared with the following syntax:

```
algo
1 var s:string[n]
```

Constant strings can be defined with the syntax `"..."`. An empty string is denoted by `""`. A string of capacity $n$ does not necessarily contains $n$ characters. The *meaningful characters* are stored at the beginning of the string (starting from the index 0). After the last meaningful characters, string are feed up with the character `\0` (null character). We will use the function `len(s)` to indicate the number of meaningful characters in a string. Many other useful functions are used with strings:

- `concat(s,t,u)`: *concatenates* the strings `s` and `t` and stores the result in the variable `u` (it should have enough capacity to store the result). Concatenation means that the string `t` is *written* at the end of the string `s`. For example, the concatenation of `"abc"` and `"def"` yields `"abcdef"`.
- `string(expression)`: this function converts the `expression` into a string representation. For example, the string representation of a the integer `56` is the string `"56"`.

**Example 5.1 : Analyse a date**

Suppose that a string represents a date (for example `"06/12/2024"`). We want to extract the date parts (day, month and year). We suppose that the date is well-formed.

algo

```
…
const n=12 var s,t:string[n]
var i,j,l,day,month,year:integer
begin
    …
    l=len(s)
    i=0
    for j=0 to n-1 do t[i]="0"
    j=0
    while i < l do
        if s[i]>='0' and s[i]<=9 then
        begin
            t[j]=s[i]
            j=j+1
            i=i+1
        end
        else break
    day=integer(t)
    for j=0 to n-1 do t[i]="0"
    j=0
    i=i+1
    while i < l do
        if s[i]>='0' and s[i]<=9 then
        begin
            t[j]=s[i]
            j=j+1
            i=i+1
        end
        else break
    month=integer(t)
    for j=0 to n-1 do t[i]="0"
    j=0
    i=i+1
    while i < l do
        if s[i]>='0' and s[i]<=9 then
        begin
            t[j]=s[i]
            j=j+1
            i=i+1
        end
    year=integer(t)
    print(day,month,year)
end
```

## 3.2. Strings in Python

In Python, strings are treated as immutable list of characters. They can be indexed by integers and there is limit on the size of a string except the memory size. It is possible to compute the length of a string by the function `len` and to enumerate the characters together with their indices thanks to the function `enumerate`. String are also iterable just as lists.

**Remark:** Strings are objects in Python. An object is a data attached to some given behaviors called methods. To execute a method `m` of an object `obj`, we use the syntax `obj.m()`.

Strings can be built using many possibilities:

- Constant strings: characters are places within double quotes, simple quotes (for one-line strings) or between two occurrences of `"""` for multi-line strings.
- From a list: using the constructor `str` it is possible to build a string from a list of characters. For instance, the class to `str(["a","b","c"])` returns the string `"abc"`.

- Using the method `join`. Let `arr` be an array of strings and `sep` a string. The call to `sep.join(arr)` returns an new string in which the elements of `arr` are concatenated but separated with `sep`. For instance, the call to `",".join(["1","2","3"])` yields the string `"1,2,3"`.
- The f-strings. Starting from version 3.7, Python added f-strings as a very practical way to format data. An f-string starts with `f` and include braces that contain expressions. Assume `a=1` and `b=2`, then the f-string `"{a}+{b}={a+b}"` produces the string `"1+2=3"`. It is easy to see that each expression is just replaced with its value in order to build the final string.

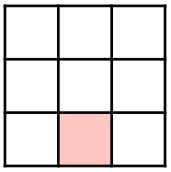
Pythons strings contains lot a useful methods, but we won't cover them in this chapter. For the moment, let's implement the date algorithm in Python (using string methods, the program will be much shorter):

```py
...
s:str=...
t:str=""
i=0
l=len(s)
while i<l:
    if '0'<=s[i]<='9':
        t+=s[i]
        i+=1
    else:break
day=int(t)

t=""
i+=1
while i<l:
    if '0'<=s[i]<='9':
        t+=s[i]
        i+=1
    else:break
month=int(t)

t=""
i+=1
while i<l:
    if '0'<=s[i]<='9':
        t+=s[i]
        i+=1
year=int(t)
print(f"{day}/{month}/{year}")
```

## 4. Multi-dimensional arrays

All arrays that we manipulated in this chapter are one-dimensional, that is, elements are simple objects and are indexed using one index. In many applications, the elements could be themselves arrays. In this case, we should provide two indices to access data, one to locate a first array of elements, and and a second one to locate the element itself. This is called a bi-dimensional array, or simple a matrix. Obviously, it is possible to define 3-dimensional arrays, 4-dimensional arrays, or higher.

Let's focus on matrices. A matrix is composed of rows, each of them is composed of columns. We use two indices to access data. If `mat` is a matrix, then `mat[i][j]` is the element in the $i$-th row and the $j$-th column. Consider a square *matrix* (a matrix whose as many rows as columns). It is represented schematically as (the pink box indicates the elements at row 2, column 1 or `mat[2][1]`):



## 4.1. Bi-dimensional arrays in the algorithmic language

Matrices are declared by defining the number of rows and columns. A matrix with `m` rows and `n` columns is declared as follows:

algo
```
1 var mat:array[m][n] of integer
```

It can be initialized to 0 by:

algo
```
1 for i=0 to m-1 do
2     for j=0 to n-1 do
3         mat[i][j]=0
```

## 4.2. Bi-dimensional arrays in Python

Matrices are just lists of list in Python. The `numpy` module support matrices and offers very interesting panoply of functions to deal with (again, this module will not be covered in this course). Assume we want to create a matrix with 3 rows and 3 columns; this is done by:

py
```
1 mat=[[0,0,0],[0,0,0],[0,0,0]]
```

In fact, there is a problem with this syntax since we need to know exactly how many rows and columns are there. We can be tented by using the syntax `[[0]*3]*3`, but this does not work as expected (why?). Although comprehension has not been introduced in this course, we will use it to define matrices. We just have to write:

py
```
1 mat=[[0]*3 for _ in range(3)]
```

Bi-dimensional arrays are often handled using nesed loops. The outer loop iterates over rows, and the inner loop iterates over columns. Obviously, complex tasks may require more elaborate loops. Consider a program that computes $\max_r \min_c A_{r,c}$, where $A$ is a bidimensional matrix. This program calculates the largest minimum value across columns. Let's consider the following example:

$$\begin{pmatrix} 1 & 4 & 5 \\ 0 & 7 & 3 \\ 7 & 4 & 8 \end{pmatrix} \quad \text{(Eq. 2)}$$

The minimum values for each row are 1, 0 and 4 respectively. In this case, the program should display the value 4 (the largest minimum value).

py
```
1  if __name__=="__main__":
2      ... # mat is initialized to some content
3      max_row=-float("inf") # float("inf") is the infinity 'value' (for all float f: f<=infinity)
4      for row in mat: # iterate over rows
5          min_col=float("inf") # initialize min_col to -infinity
6          for elt in row: # browse a column
7              if elt<min_col:
8                  min_col=elt
9          if max_row<min_col: #update the max_row variable
10             max_row=min_col
11     print(max_row)
```

## 4.3. Enumerating arrays and unpacking in loops

Looping over indices and values may be useful in many situations. Consider a program that determines the index of the greatest element in an array.

```py
arr=[...]
max_elt=-float("inf")
ind=-1
for i in range(len(arr)):
    if max_elt>arr[i]:
        max_elt=arr[i]
        ind=i
print(f"The largest value is at index {ind}")
```

There is nothing wrong with this code. Nonetheless, we had to specifically use the length of the array and using indices to loop over the array's values. It would be more interesting if we could loop over both indices and elements. That's what `enumerate` is meant to. Both indices and values can be browsed using the `enumerate(arr)` function. At each iteration, it returns a tuple containing two elements: the current iteration's index and the corresponding value. The previous code can be rewritten as:

```py
arr=[...]
max_elt=-float("inf")
ind=-1
for i,v in enumerate(arr):
    if max_elt>v:
        max_elt=v
        ind=i
print(f"The largest value is at index {ind}")
```

Note the use of unpacking to get the values of the index and the value in two separate variables.

Unpacking can be further used with arrays and tuples. For instance, assume `list` is an array of tuples: `[(2,5),(6,7),(1,0),(5,7)]`. This list can be browsed in several ways (for instance, `for t in arr`). However, unpacking can yield a more readable version:

```py
for i,j in arr:
    # do something
```

In this code, we are iterating over the values of `arr`. In each iteration, the elements of the array (actually they are tuples) are unpacked into two variables `i` and `j`. In general, this will make programs easier to write and to understand.