AIMS African Institute for Mathematical Sciences CAMEROON

1. Python

1.1 Introduction to algorithms

Kevin Jeff FOGANG FOKOA

AIMS Cameroon

kevin.fogang@aims-cameroon.org

Reading Week 2025/2026

Plan

1. Variables and Assignments
2. Instructions
3. Structure of an Algorithm
4. Functions and Procedures
5. Data Structures

Some definitions

Instruction:

Algorithm:

Program:

Software (application):

Pseudo-code:

Comment:

Some definitions

Instruction: It is a command that can be understood and executed by a ”computer”.

Algorithm: It is a finite sequence of instructions and ordered rules that allows a very specific task or problem to be solved.

Program: Is an algorithm translated into a programming language.

Software (application): It is a set of programs and data that tell a ”computer” how to perform specific tasks.

Pseudo-code: Is a conventional language (almost natural language) that a computer science community uses to create/write algorithms.

Comment: portion of code that is not executed by the ”computer”. It begins with a ”//”.

Variable and Constant

Variable: Is a named memory location (identifier) that allows data stored there to be manipulated.

Types of variables:

- Integer. Example : 4, 8, 100, -100, -15
- Real. Example : 0.0004, 5, -1, -0.5
- Boolean. true and false
- String. Example : ”a”, ”hi”, ”202”, ”15.5”, ”AIMS Cameroon”

Constant: This is a variable that does not change during the execution of the algorithm. For example: an integer variable to manipulate the months of the year will always have the value 12. 24 to manipulate the hours in French format.

How to write variables

Convention to respect:

- The first letter of the identifiers must not be a number (0 to 9), accented character (é, è, à, etc.) or special character. (&, %, $, -,_, etc).
- Avoid special and accented characters in variable names.
- Constant identifiers are usually words in uppercase.
- Variable names should be understandable: avoid crazy abbreviations.

NB : In practice (programming), you should always refer to the programming language documentation to know what is allowed and what is not.

Declaration and initialization of variables

To declare a variable, we write as follows: **type : name**. Example:

- Let's declare a variable to store the age. **integer : age**
- Let's declare a variable to store the registration number. **string : mat**
- Let's declare a variable to store the grade. **real : grade**

type : name $\leftarrow$ **value**. Example:

- Let's declare a variable to store the age. **integer : age := 20**
- Let's declare a variable to store the registration number. **string : mat := "24L35"**
- Let's declare a variable to store the grade. **real : grade := 20**
- Let's declare a variable for the number of months. **constant integer : MONTHS := 12**

Assignments

Assignment is an instruction through which a value is assigned to a variable.
Variable initialization is therefore an assignment operation.
The two syntaxes generally used to denote an assignment are as follows:

variable_name := value

or

variable_name ← value

We write one instruction per line, and we can optionally mark the end of an instruction with a ”semicolon” (;).

If we want to write multiple instructions on the same line, we must separate them with ”;”. Example:
integer : age := 0; string : mat

The different types of instructions:

- Assignment instruction (Already covered! :)
- Input and output (write and read) instructions
- Arithmetic and comparison instructions
- Conditional instructions
- Loop instructions

Instructions of reading and writing

Output (write): It is an instruction through which the program is connected to the screen to display a message or the value contained in a variable. Its syntax is as follows: **write(variable_name)**.
If we want to tell the machine: display on the screen the value contained in the variable ”age”, then we would write:

write(age)

Input (read): It is an instruction through which the program is connected to the keyboard to capture the value (event) that will be entered by the user in order to store it in a variable. Its syntax is as follows: **read(variable_name)**.
If we want to tell the machine: take the value entered by the user and store it in the variable ”age”, then we would write:

read(age)

Arithmetics and Comparisons Instructions

Arithmetic: These are instructions that allow performing arithmetic operations. To do this, we use arithmetic operators: +; -; /; *; % (modulo); div (integer division). The result of an arithmetic instruction is a number. Example:

c := a % b // Stores the remainder of the division of a by b in c

Comparisons: These are instructions that allow performing comparison operations. To do this, we use comparison operators: $<$; $>$; $<=$; $>=$; ==; $!=$ or $<>$ (different). The result of a comparison instruction is a boolean. Example:

bool := 10 == 20 // boul contains the value ”False”.

Conditional Instructions

These are instructions that execute when the condition is met. Their syntax is as follows:

---

if condition **then**
  instructions...
end if

---

if condition **then**
  instructions...
else
  instructions...
end if

Conditional Instructions

These are instructions that execute when the condition is met. Their syntax is as follows:

---

if condition **then**
  instructions...
end if

---

if condition **then**
  instructions...
else
  instructions...
end if

Exemple : Delta of Kronecker

if i == j **then**
  write(1)
else
  write(0)
end if

Loop Instructions

for loop: *i* is the counter that helps determine the current step. *begin* and *end* are the initial and final values of the counter, and *s* represents the step (it can be negative or positive).

while loop: The instructions inside this loop are executed as long as the condition is satisfied.

repeat loop: The instructions inside this loop are executed first, then they are re-executed as long as the condition is valid.

Loop Instructions

for loop: *i* is the counter that helps determine the current step. *begin* and *end* are the initial and final values of the counter, and *s* represents the step (it can be negative or positive).

while loop: The instructions inside this loop are executed as long as the condition is satisfied.

repeat loop: The instructions inside this loop are executed first, then they are re-executed as long as the condition is valid.

```
for i := begin to end step s do
    Instructions...
end for
```

Loop Instructions

for loop: *i* is the counter that helps determine the current step. *begin* and *end* are the initial and final values of the counter, and *s* represents the step (it can be negative or positive).

while loop: The instructions inside this loop are executed as long as the condition is satisfied.

repeat loop: The instructions inside this loop are executed first, then they are re-executed as long as the condition is valid.

```
for i := begin to end step s do
    Instructions...
end for
```

```
while condition do
    Instructions...
end while
```

Loop Instructions

for loop: *i* is the counter that helps determine the current step. *begin* and *end* are the initial and final values of the counter, and *s* represents the step (it can be negative or positive).

while loop: The instructions inside this loop are executed as long as the condition is satisfied.

repeat loop: The instructions inside this loop are executed first, then they are re-executed as long as the condition is valid.

```
for i := begin to end step s do
    Instructions...
end for
```

```
while condition do
    Instructions...
end while
```

```
repeat
    Instructions...
until condition
```

Indentation

Indenting an algorithm or a text in general means aligning it properly.

Indentation helps with the understanding and readability of the algorithm (the same applies to source code). It is always preferable, if not essential, to indent an algorithm (or code).

General structure of an algorithm

Algorithm: Name of the algorithm

    Variable declarations
    Variable initialization

begin

    Instructions...

end

Example

Write an algorithm that sum up two numbers.

Algorithm: Sum of 2 numbers;

Variables:

Integer : x, y, sum
sum := 0; x := 0; y := 0

Begin

write(”Please enter the first number:”)
read(x)
write(”Please enter the second number:”)
read(y)
sum := x + y
write(a, ” + ”, b, ” = ”, sum)

End

Effectiveness, Correct

An algorithm is more efficient (performing) than another if the execution time of its worst case is lower (an order of magnitude term) than (of the unfavorable case) of the other. An efficient algorithm consumes less hardware resources compared to those that are not.

An algorithm is said to be correct when for each instance of input data, it returns correct results.

A result is correct if it is expected (expected with respect to the input data).

Function

Function: A sub-algorithm that accepts parameters and local variables of its own, and returns a result after execution.

Function function_name(type1:param1, type2:param2, ..., typeN:paramN)

    Local variable declarations
    Local variable initialization

Begin

    Instructions...

    Return result

End

Procedure

Procedure: A sub-algorithm that accepts parameters (may not have parameters) and local variables of its own, and does not return a result after execution.

Function function_name(type1:param1, type2:param2, ..., typeN:paramN)

    Local variable declarations
    Local variable initialization

Begin

    Instructions...

End

Example

Write an algorithm that print the maximum between two numbers.

Example

Write an algorithm that print the maximum between two numbers.

Function max(Int:a, Int:b)

Begin
    if $a >= b$ then
        return a
    else
        retourn b
End

Example

Write an algorithm that print the maximum between two numbers.

Function max(Int:a, Int:b)

Begin
    if $a >= b$ then
        return a
    else
        retourn b
End

Algorithm: Maximum between two numbers;

    Interger : x, y

Begin
    write(”Please enter the first number:”)
    read(x)
    write(”Please enter the second number:”)
    read(y)
    write(”The maximum is ”, max(x, y))
End

Formal and effective Parameters

Formal parameters (also called parameters) are the variables given when we define a function. While the effective or actual parameters (also called arguments) are the values, variables or functions which are passed to a function when it is invoked.

In our previous example, the formal parameters are: a and b; And the effective ones are x and y.

Parameters Passing

- Pass by value: The actual parameter is copied into a local variable, and it is the latter that is used to perform the calculations in the called function. Therefore, modifying the local variable cannot modify the one outside the function.
- Pass by reference: The actual parameter provides direct access to the original variable. There is no copy here. Therefore, any modification of the variable within the called function results in the modification of the variable passed as an argument.

NB : Depending on the programming language, it's important to know which types of parameter passing are supported.

Data Structures

Data structures are abstract objects that denote the way data will be organized and stored in order to process them (search and modification) more easily. As data structures, we can cite:

- Simple, primitive or basic data structures (boolean variables, integers, reals, etc.; constants)
- Records
- One-dimensional and multi-dimensional arrays
- Queues and stacks
- Linked lists
- Trees
- Graphs

Records

Records allow you to define compound types, that is, types that have variables of different types.

Records

Records allow you to define compound types, that is, types that have variables of different types.

Syntax :

name_rec = Records
    name_var1: type1;
    name_var2: type2;
    .
    .
    name_varN: typeN;
End_record

Records

Records allow you to define compound types, that is, types that have variables of different types.

Syntax :

name_rec = Records
    name_var1: type1;
    name_var2: type2;
    .
    .
    name_varN: typeN;
End_record

Example : Create a PC type characterized by brand, clock frequency and RAM size.

PC = Records
    brand: string;
    freq: real;
    ram: interger
End_record

Arrays

This is a data structure whose elements are all of the same type and accessible via indexes.

name_array = Array[size_array] of ”name_type”

Example 1: Create an array that can contain the ages of 100 people
age_tab = Array[100] of integers

Example 2: Create an array that can contain characteritics of 100 computers
carc_tab = Array[100] of PC