Chapter 2

Simple sequential algorithms

1. Introduction

As stated in chapter 1, algorithms correspond to procedures that solve a well-defined and solvable problems. Algorithms are well-formed descriptions of how to solve the problem but are not designed to run on a specific computer. In contrast, programs are a precise procedures to solve problems that are especially designed to run on computers. Still a program, depending on many criteria, can require specific conditions in order to be executed.

There are plenty of techniques to build algorithms and programs. Some are easy to build by humans while being difficult to maintain after deploying, others can be a bit harder to apprehend but can be very robust and very suitable for maintenance. Among the simplest ways to define an algorithm, we can just describe the steps to follow in order to solve a problem. Let's imagine that we want to prepare a cake. How can we describe the procedure to make that. In general, cooking books propose recipes in terms of sequential steps, i.e., a list of steps that should be applied in a strict order. For example, one can make a cake by:

Example 2.1 : How to cook a cake *sequentially*

1 – Preheat the oven.
2 – Mix butter and sugar together in a bowl.
3 – Add eggs, one at the time and briefly beat.
4 – Mix in vanilla.
5 – Combine flour and baking powder in a separate bowl.
6 – Incorporate wet ingredients and mix well.
7 – Pour the batter in a cake pan.
8 – Bake in the oven for 30 minutes.

This kind of algorithm is referred to as *sequential* because it is executed one step at a time, following a specific order. In the above example, the cake cannot be properly baked if the recipe is not followed progressively from steps 1 to 8. If one starts the recipe at step 3 or skips from step 2 to step 5, the cake may not bake properly.

When algorithms are created in this manner, if they are correct, their execution is usually fast. It can even be made faster by using several CPUs. However, it is not always simple to create algorithms in this way; one needs to be an expert in order to get things working. This is the most used way to build programs, and this the programming model that will be used throughout this course.

A different approach for creating algorithms is to provide the properties of the objects that may be used to solve the problem. Let's reconsider the cake recipe. We know that a cake is prepared with flour, eggs, sugar, butter, and vanilla,

and we need an oven. We know that an oven can be heated to a specific temperature, and that each ingredient has a minimum degree of cooking and a maximum degree at which it will burn. We also know that flour is dry and cannot be mixed directly with sugar (otherwise, the cake would taste bad). Eggs should not be incorporated immediately with flour. Once all logical properties have been established, the cooker will look for a sequence of steps that will allow the cake to be baked. Computers are very efficient at dealing with this kind of problems (one program may solve many problems of this kind). We refer to this type of programming as *declarative*. It is simpler to create since we only must specify the required objects and their properties. However, execution is often slower than sequential programming.

2. Concept of language and algorithmic language

Communication occurs when information need to be shared. When two individuals (say A and B) wish to share information, they should communicate since they don't share the same brain (figure 2.1). A encodes its information according to specific rules and then sends the message to B over a medium (air, wires, wifi, etc.). This one decodes the message to retrieve the information, following the same rules as A. This scheme is employed in all communication between distinct entities (humans, animals, and machines). We commonly refer to the rules of communication as a *language*.

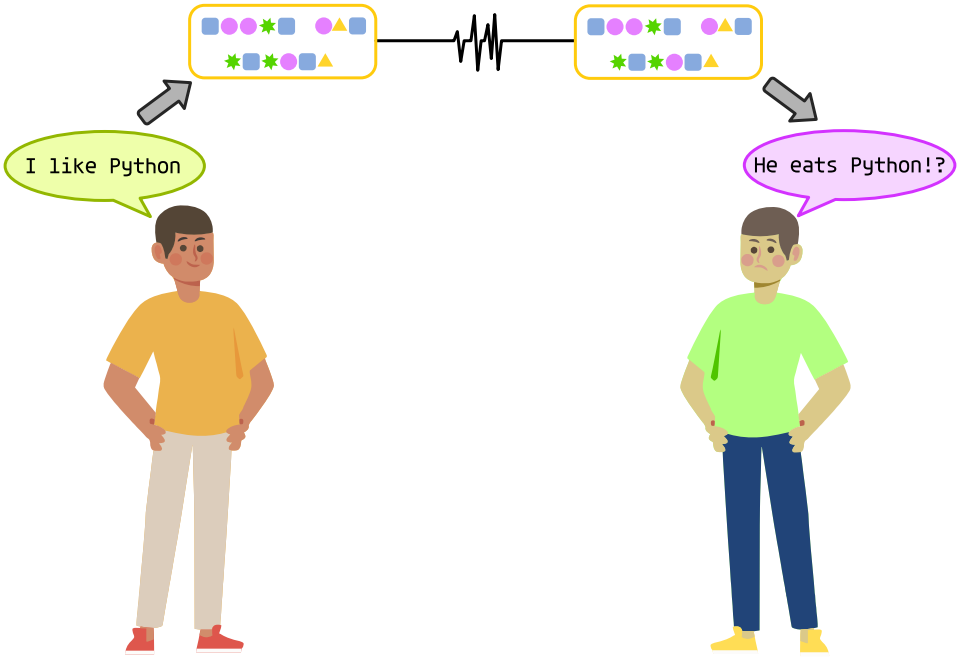


Figure 2.1 - scheme of communication

Computers are made up of electronic components that represent data in binary (on the basis of bits[1]. Every CPU has a set of basic instructions, which are coded with 0s and 1s. As a result, creating programs requires writing awful bit sequences. This is what happened with the first generation of computers. The languages of these generation are ref to as *first generation languages*.

Humans rapidly figured out that such an approach is inefficient for building advanced applications and severely limits computer commercialization. Thus, second-generation languages have been developed. To express fundamental operations in these languages, mnemonic labels have been used. Assembly language is one such languages (actually there are many assembly languages). The code below is part of an assembly program.

```
mov $1, %rax
mov $1, %rdi
mov $message, %rsi
mov $15, %rdx
```

Assembly languages are hard to use, hence they are not used by any programmer. However, they are still commonly used to create low-level programs (programs that interact directly with hardware).

[1]A bit is a data unit that can only hold 0 or 1

Languages grow increasingly human-centered with each successive generation (3rd, 4th and 5th). Many languages have been designed as imitations of English. The most notable were and continue to be: Cobol, Fortran, Pascal, C, C++, Java, Python, Javascript, SQL, etc. These languages allow programmers to submit instructions to machines using English-like constructs. However, a machine requires a special program, known as a *compiler*, to translate programs into the binary language.

2.1. Why not natural language?

Programming languages are too artificial, harsh, and emotionless for people to use as a communication tool by people. Natural language refers to daily human language (such as English, Arabic, French, Chinese, and sign languages). Faced with the extreme difficulty of employing first generation languages, the idea of using mnemonic constructions to program computers emerged, but there was some skepticism: computers were unable to understand English (at least at that the time). That is why programming languages were and continue to be very synthetic.

However, as artificial intelligence (AI) evolves, more complex machines are developed. In 2023, ChatGPT revolutionized the field of natural language processing. We may now ask intelligent robots to perform tasks simply using normal language. So, why don't we employ natural language to program computers? First, let's recall the main difference between computer languages and natural languages: formality. Computer languages are formal, which means that each construct in a program has only one meaning. This is not true for natural languages; one sentence can have several interpretations depending on the context. Consequently, natural languages are inherently ambiguous. Despite AI sophistication, it is hard to get rid of subjectivity. Artificial synthetic languages are still required at today's technological level.

In this course, we will two languages: algorithmic language and Python. Algorithmic language is a platform-independent language that allows describing solutions in a formal way. It is an abstract language, i.e., it cannot be executed on computers (unless a dedicated compiler is designed). On another hand, Python is +3rd programming language that can be executed on a large number of computers.

2.2. The algorithmic language

This is not a real programming language. It is just a neutral approach for building algorithms that are independent of the underlying hardware or software. In practice, it is difficult to build an algorithmic language that reflects all of the existing computer languages since they are so different. One such difference is the way of dealing with data types. Some programming languages are strongly typed, which implies that every piece of data should be properly associated with its type (we will talk about data types later). Other programming languages are weakly typed, which means a programmer can manipulate data without knowing its type.

In this course, the algorithmic language will be strongly typed. The instructions for this language will be introduced gradually.

2.3. The Python language

Python is an interpreted, high-level, general-purpose programming language. Currently ranked ahead of C, C++, Java, and Javascript, it is the most widely used programming language as of 2024. It is a ubiquitous language, which means it may be used in a wide range of fields, from simple administration tasks to database applications, web applications, and GUI applications. While Python can be used for building mobile applications, the language is not particularly mobile-friendly. However, it is employed in microcontrollers. Python is also known for its readability, which means that humans can easily read and understand Python programs.

Python is acclaimed for its tremendous ecosystem: the set of extensions that allow applying Python in all imaginable domains. One such domain is data analysis and artificial intelligence.

Guido van Rossum developed Python in the late 1980s. The first version, 0.9.0, appeared in 1991. Since then, Python has undergone several significant modifications, the most notable of which being versions 2.x (released in 2000) and 3.x (released in 2008). Version 3.13 is the most recent, as of September 2024. In this course, we will be using version 3.10. Installing the most recent versions of a language is usually exciting but compatibility concerns must be considered. For example, many Linux distributions rely heavily on Python for a variety of activities, and the migration can sometimes result in unexpected behavior.

Python is generally portrayed with a snake, yet the word has nothing to do with python snakes. The name originates from the British comedy group "Monty Python", whose show ran on the BBC from 1969 to 1974.

Remark: Python is weakly typed, in contrast to the algorithmic language we are employing. Python does, in fact, support type declaration, although it is optional. In this course, we will follow the convention of declaring the type of each data as much as possible.

Python syntax is richer, and it will be presented gradually. It should be noted that this is an introductory programming course. Only instructions covering the specified subjects will be provided.

3. The structure of algorithms and programs

3.1. The structure of an algorithm

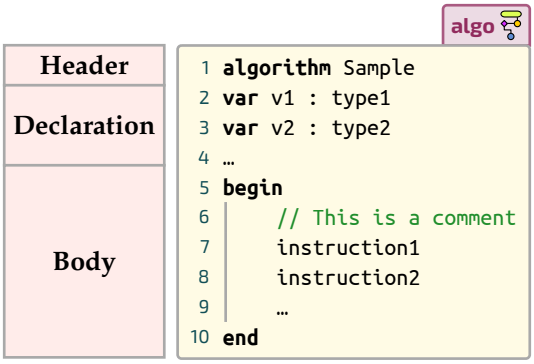
Knowing the components of an algorithm makes it easier to develop them. Algorithms can range from very simple to quite sophisticated and complex. In this last case, we often divide an algorithm into smaller parts to reduce the complexity. The basic structure of an algorithm consists of three elements:
- Header: this part is used to set the name of the algorithm. It may have additional elements that will be discussed later.
- Declaration : it describes the used variables in the algorithm
- Body : it contains a sequence of instructions for solving the problem.

The header part (line 1) begins with the keyword **`algorithm`** followed by the name of the algorithm.

The declaration part (line 2 to 4) gives the list of used variables along with their types. Each declaration begins with the reserved word **`var`** followed by a list of names, a colon, and then the type of the variables.

The body part (line 5 to 10) which begins with the keyword **`begin`**, contains a sequence of instructions (usually one instruction per line) and terminates with the reserved word **`end`**. A comment starts with the characters `//`.

The section from the keyword **`begin`** to the keyword **`end`** is called a *block*. The body part contains at least one block. It is, therefore, possible to have any number of blocks, one inside another.



We can notice spaces at the beginning of lines 6–9. These spaces indicate that the instructions belong to the same block. We call them *indentations*. The rule of indentation is simple: each time a block is created, extra spaces are added. When the block ends, the extra spaces are removed. Indentation is optional in this language (as it is in C, Java, and many other languages), although it is an essential requirement for code quality. A properly indented code is simple to understand and maintain. We will continue to use indentation in algorithms created in this course (see the next section to know why).

It is kind of tradition to write a special program for every new language, or when we learn a new one. The program simply prints the expression `Hello world`, and is generally used to test whether the compiler has been correctly set up.

algo
```
1 algorithm HelloWorld
2 begin
3     print(“Hello world!”)
4 end
```

3.2. Introduction to Python philosophy

Python is a kind of odd language. In contrast to the algorithmic language, Python does not have a general structure. This does not mean that Python programs are badly organized. Python programs can be highly organized and very easy to read.

First, the program name is defined within the file name that contains the program. For instance, the previous `HelloWorld` algorithm will correspond to the file `HelloWorld.py`. It is possible to define variables whenever we want, but it is essential to maintain the program as simple as possible. The same is true for the instructions. There is no clear body part, although it is possible to specify a main part of a program. Let's consider the following HelloWorld Python program:

py
```
1 # Here we go
2 if __name__=="__main__":
3     print("Hello world")
```

Comments start with the character # and last until the end of the line. One important feature of Python is that there are no begin or end keywords; rather, indentation is mandatory. In the above code, the block starting is indicated by :, and an extra indentation is required if we go to a new line. This is how Python's code remain easy to read. The body of the program begins on line 3.

Remark: Throughout this course, the algorithmic language and Python will be used in conjunction. Note, however, that Python programs often differ from those written in other languages. In reality, programs written in the algorithmic language can be simply translated into a language like C. Making the same thing using Python will provide unconventional Python programs that may take longer to execute. The term *Pythonic* code refers to the fact that the program is written in accordance with Python's principles and established practices.

4. Data: variables and constants

A program needs memory to be executed since the number of register on the CPU is very limited. Memories are addressed using integers (coded on 8 bits, 16 bits, 32 bits, 64 bits, or more according to the generation of microprocessors). In fact, the memory is composed of cells, each of which corresponds to a byte (8 bits).

In the first and second generations of languages, programmers were required to handle memory directly by manipulating the addresses at which data was stored. Things become more difficult when different data types need different sizes and coding. Third-generation languages solved the problem by using mnemonics to refer to data rather than addresses. A variable is a container with a name that stores its value in a known location in memory. Any variable has:

- A name: a sequence of letters, digits, and possibly special characters. A name should start by a letter or the character _ followed by a sequence of letters of digits or the symbol _. Note that this one has a special meaning in Python. A variable name should be unique in a program. There is a difference between capital letters and small letters.
- A type: the type of a variable corresponds to all possible values that it can take. The type defines not only the required amount of memory to store data but also the rules that are applied to it.

Basic types are given in the following table:

Type	Algorithm	Algorithmic examples	Python	Python examples
Booleans	`boolean`	`true, false`	`bool`	`True, False`
Integers	`integer`	`0, -5,42`	`int`	`0, -5,42`
Characters	`char`	`'a', '0', '\n'`	`str`	`'a', "v"`

Strings	`string`	`"example"`	`str`	`"example",'example'`
Reals	`float`	`4.5, 0.5, 1e-3`	`float`	`4.5, 0.5, 1e-3`
Null	-	-	`NoneType`	`None`

In reality, `string` is a complex type and is very similar to arrays (covered in a later chapter). However, they are regularly used, especially in input/output operations. Hence, they are partially covered in this chapter. Python strings can be built in many ways; they may start and end with `'` or `"` (they are called delimiters). In both cases, this is a one-line string. Python also has a multi-line string delimiter written as `"""`. A string can store any number of characters.

A variable is a container that stores a value. It is considered mutable if its value may be changed (like with integers, floats, and booleans). Immutable variables can only have an initial value and cannot be altered thereafter. In Python, strings are immutable. In the algorithmic language, variables are declared as shown in the example below:

algo
```
1 var i,j : integer
2 var s : string
```

In this language, the type of each variable should be declared and cannot be changed. Using an undeclared variable or using a variable in the wrong way will raise a compilation error.

Declaring variable types is optional in Python. However, it is good programming practice to define the types of variables. The above example might be written in Python as:

py
```
1 i:int
2 j:int
3 s:str
```

Naming conventions

Despite there is no obligation about naming variables, different conventions exist in programming. It is generally better to have meaningful names, which reflect what the variable is intended to contain. This can include one or more words that should be joined by one of two main conventions:

- Camel cases: all words begin with a capital letter. However, the first word can or cannot begin with a capital letter (pure Camel case vs. Pascal case). An example of such a name is `studentAge`. This convention will be used in the algorithmic language.
- Snake case: - Snake case: all words are in small letters, separated by the character _. An example of such a name is `student_age`. This convention is used in Python.

It should be noted, however, that very long variable names are not recommended since they reduce program readability. Variables can have a limited lifespan in some situations. In this case, they can have simple names like `i` or `x`.

Nevertheless these lines do not define the variables; they are only used to denote their types. This is particularly helpful when using many IDEs (Integrated Development Editors) that can offer assistance dependently on the types of variables. When a variable has a value assigned to it, it is effectively created. One common feature of weakly typed languages is the ability for variables to change type. In Python, this can be done by assigning a different value to a variable (although this is not recommended).

It should be noted that if a variable is unknown or utilized improperly a runtime error will be triggered. We will keep using the warning ⛔ about **fatal errors** that one should absolutely avoid when writing programs.

⛔

Never forget to assign an initial value to a variable. Although Python does not support uninitialized variables, many strongly typed languages allow that.

Constants are a useful means to define immutable values in a program. Consider a method for calculating the average mark of N students. The mathematical formula for average is identical for all possible values of N. In reality, N is not a variable; it is just information that will assist us in calculating the average.

In the algorithmic language, constants have a type and a value that cannot be changed (attempting to do so results in a compilation error). This can be done using the following syntax (to be placed into the declaration part):

algo
```
1 const a=8
2 const s="a constant string"
```

Python has no constants in the strict sense. We can assign values to a name, but this is a variable that can be changed at any time. However, there is a convention for naming variables that should be viewed as constants (by committed developers). This convention involves using a name with just capital letters, sometimes separated by the character _. The following example shows some constants (that are not actually constants):

py
```
1 VAT_RATE:float=19.6
2 HELLO_MSG:str="Hi there!"
```

Note that characters are basically from the ASCII codes. This is a set of 256 symbols that represent the basic characters on a computer system. Each character has a code. For instance, the code of `"A"` is 65, and the code of `"B"` is 66. As we can see, the codes of capital letters succeed one another. This is also the case of small letters and digits as well. The code of a character `c` can be retrieved by the function `ord(c)` whereas the character associated with a code `i` is given by the function `chr(i)`. The following table gives the 256 characters of the ASCII codes:

0	1	2	3	4	5	6	7	8	9
[Null character]	[Start of Header]	[Start of Text]	[End of Text]	[End of Trans.]	[Enquiry]	[Acknowledgement]	[Bell]	[Backspace]	[Horizontal Tab]
10	11	12	13	14	15	16	17	18	19
[Line feed]	[Vertical Tab]	[Form feed]	[Carriage return]	[Shift Out]	[Shift In]	[Data link escape]	[Device control 1]	[Device control 2]	[Device control 3]
20	21	22	23	24	25	26	27	28	29
[Device control 4]	[Negative acknowl.]	[Synchronous idle]	[End of trans. block]	[Cancel]	[End of medium]	[Substitute]	[Escape]	[File separator]	[Group separator]
30	31	32	33	34	35	36	37	38	39
[Record separator]	[Unit separator]	[Space]	!	"	#	$	%	&	'
40	41	42	43	44	45	46	47	48	49
(	)	*	+	,	-	.	/	0	1
50	51	52	53	54	55	56	57	58	59
2	3	4	5	6	7	8	9	:	;
60	61	62	63	64	65	66	67	68	69
<	=	>	?	@	A	B	C	D	E
70	71	72	73	74	75	76	77	78	79
F	G	H	I	J	K	L	M	N	O
80	81	82	83	84	85	86	87	88	89
P	Q	R	S	T	U	V	W	X	Y
90	91	92	93	94	95	96	97	98	99
Z	[	\	]	^	_	`	a	b	c
100	101	102	103	104	105	106	107	108	109
d	e	f	g	h	i	j	k	l	m
110	111	112	113	114	115	116	117	118	119
n	o	p	q	r	s	t	u	v	w
120	121	122	123	124	125	126	127	128	129
x	y	z	{	|	}	~	[Del]	Ç	ü

130	131	132	133	134	135	136	137	138	139
é	â	ä	à	å	ç	ê	ë	è	ï
140	141	142	143	144	145	146	147	148	149
î	ì	Ä	Å	É	æ	Æ	ô	ö	ò
150	151	152	153	154	155	156	157	158	159
û	ù	ÿ	Ö	Ü	ø	£	Ø	×	ƒ
160	161	162	163	164	165	166	167	168	169
á	í	ó	ú	ñ	Ñ	ª	º	¿	®
170	171	172	173	174	175	176	177	178	179
¬	½	¼	¡	«	»	░	▒	▓	│
180	181	182	183	184	185	186	187	188	189
┤	Á	Â	À	©	╣	║	╗	╝	¢
190	191	192	193	194	195	196	197	198	199
¥	┐	└	┴	┬	├	─	┼	ã	Ã
200	201	202	203	204	205	206	207	208	209
╚	╔	╩	╦	╠	═	╬	¤	ð	Ð
210	211	212	213	214	215	216	217	218	219
Ê	Ë	È	ı	Í	Î	Ï	┘	┌	█
220	221	222	223	224	225	226	227	228	229
▄	¦	Ì	▀	Ó	ß	Ô	Ò	õ	Õ
230	231	232	233	234	235	236	237	238	239
µ	þ	Þ	Ú	Û	Ù	ý	Ý	¯	´
240	241	242	243	244	245	246	247	248	249
≡	±	‗	¾	¶	§	÷	¸	°	¨

Note that some special characters can be written using special syntax, called escape sequence. The following list is not exhaustive.

- `\t`: known as tabulation, it is used to organize data and making indentation.
- `\n`: used to represent a new line.
- `\r`: used to represent a carriage return (in Windows, new lines correspond to the sequence `\r\n`).
- `\\`: used to represent the backslash itself (\).
- `\'`: used to represent the simple quote ‘.
- `\"`: used to represent the simple quote “.

5. Basic operations

Variables and constants are great, but they do not allow for much fun. Magic happens when they are combined. The expression is the core component of computation in every programming language. An expression can be composed of variables, constants, and operations (also known as operators).

Operations basically correspond to arithmetic and logical operations. Nevertheless, programming languages provide a far wider range of operations. One basic example of an operation is the addition (denoted by the symbol +). In the expression `x+y`, `x` and `y` are called the arguments of the operation. This is a *binary* operation, as it takes two arguments. Some operations are *unary*, meaning they only require one argument.

An operation is typically characterized by its arity (the number of arguments required), the type of used arguments, and the type of result. The table below shows the main arithmetic and logical operations in both the algorithmic language and Python.

Operation	Arity	Algorithm	Python	Arguments' types	Result
Addition	Binary	+	+	`integer` or `float`	`integer` or `float`
Subtraction	Binary	-	-	`integer` or `float`	`integer` or `float`
Minus	Unary	-	-	`integer` or `float`	`integer` or `float`

Multiplication	Binary	*	*	integer or float	integer or float
Division	Binary	/	/	integer or float	float
Integer division	Binary	div	//	integer	integer
Modulus	Binary	mod	%	integer	integer
Power	Binary	^	**	integer or float	integer or float
Comparison	>=2	>,>=,...,==,!=	>,>=,...,==,!=	Any	boolean
Conjunction	Binary	and	and	boolean	boolean
Disjunction	Binary	or	or	boolean	boolean
Negation	Unary	not	not	boolean	boolean

Never try to make a division or a modulus if the denominator is zero.

5.1. Expression evaluation

Evaluating an expression means computing its value. For instance, the expression `5+2.7` yields `7.7`. When an expression involves variables, then the expression value is computed by substituting each variable with its value. Suppose `x` is associated with the value `5.5` and `y` is associated with the value `2`, then the expression `x*y` yields `11.0`.

In general, expressions are evaluated left to right. For example, `x+y+2` is evaluated by computing `x+y` first, then the rightmost `+`. When an expression contains many operations, the following rules are applied:
- Evaluations are performed left to right for the same operation.
- Unary operations have the highest priority after the power operation.
- Multiplicative operations (`*`, `/`, `//`, `%`) have higher priority than additive operations (`+`, `-`).
- Logical operations are next, with the priority going top to down: `not`, `and`, `or`.
- Priorities can be changed using parenthesis.

For instance, the expression `(3+2)*5**3` is computed with the following steps:
1 – $3 + 2 = 5$
2 – $5^3 = 125$
3 – $5 * 125 = 625$

6. Basic instructions

6.1. Assignment

The assignment is the simplest instruction in a program. It means associating a new value to a variable. The syntax is roughly the same in the algorithmic language and Python `name=expression`. This instruction is executed as follows:
1 – Evaluate `expression`, let v be its value.
2 – Associate `name` to the value v.

There are several Python assignment variations that can reduce the length of the written code. The assignment `i=i+expression` can be expressed as `i+=expression`. This *syntactic sugar* works with any other binary operation. This is not the most notable way in which Python excels. In fact, assignment is more powerful than just associating a value to one variable. For instance, it is possible to assign many values to many variables at the same time. The instruction `i,j=u,v` means that `u` is assigned to `i` and `v` is assigned to `j` (this instruction is known as *unpack*). If we are not interested in the value of v, then we can just use _ instead of j. Note that it is also possible to write `i=u,v` but this corresponds to a more complex data type that will be covered later. We should be careful with the number of variables and expressions; the number of variables should never be greater than the number of expressions.

One straightforward consequence of multi-assignment is how easy it is to *exchange* the values of two variables `x` and `y`. In Python, this is simply done by:

```py
x,y=y,x
```

However, in the algorithmic language, we should use a third variable (let's call it `tmp`):

```algo
tmp=x
x=y
y=tmp
```

It is also possible to make sequential assignments with `v1=v2=v3=expression` which assign the value of `expression` to `v3`, then the value of this one to `v2`, then value of this one to `v1`.

6.2. Input and output

A program uses input and output actions to communicate with its *environment*. Input refers to any means that allow a program to receive data, such as a keyboard, mouse, file, other programs, etc. Output refers to any mechanism by which a program can send data, such as a screen, printer, file, other programs, etc. In computers, the keyboard and screen are usually known to as standard input and output.

In the algorithmic language and Python, input will be done by the function input(s), where `s` is a constant string that will be displayed on the screen. Suppose a program is executing `a=input("what is the VAT?")`, the user will see the text `what is the VAT` on the screen, and the program execution is suspended. The user may type some characters and validate them by typing the enter key. Suppose that the user typed the keys `19.6` and then the enter key. This will assign the string `"19.6"` to the variable `a`. In this case, it is not possible to make arithmetic operations with `a`. First, it should be converted to the right type using the function `float(...)`. The conversion to integers is made by the function `int(...)`. Note that the conversion may trigger an error if the string is not a number. We will see later how errors are handled.

The standard output is made by the function `print(s,t,...)`. This function prints its arguments (we can pass any number of arguments with any type). It will print all of its arguments on the screen, then insert a new line. For instance, suppose the variable `a` equals 1, then the call `print("a=",a)` will print `a=1` on the screen.

6.3. The *pass* statement

Python includes a special statement that does nothing called `pass`. This instruction is required in a block that has no action due to the indentation used by Python. In some cases, this instruction serves as a placeholder for code to be written later.

7. Constructing a simple program

As an illustration of all of the concepts covered in this chapter, we will develop a program that asks the user for the name and price of a product, then outputs the net price (price without tax) of the product. We assume that the VAT rate is fixed at 19.6%.

To write the program, we first need to know the input data and their types. In this case, we need the name of the product (string) and its price (a non-negative float). Next, we need to define its output: the net price of the product (float). The program will use a constant to define the VAT rate. To compute the result, we know that $p = np + vat * np = np * (1 + VAT)$. We, hence, obtain $np = \frac{p}{VAT+1}$.

The algorithmic version is given here:

algo

```
algorithm INVERSE_VAT
var price,net_price:float
var product_name:string
const VAT_RATE=0.196
begin
    product_name=input(“What is the product name? “)
    price=float(input(“What is its price in dinars? “))
    net_price=price/(1+VAT_RATE)
    print(“The net price of”,product_name,"is",net_price,"Dinars")
end
```

The Python version is given here (the output needs some formatting):

py

```
if __name__=="__main__":
    VAT:float=0.196
    product_name:str=input("What is the product name? ")
    price:float=float(input("What is its price? "))
    net_price:float=price/(1+VAT)
    print("The price of",product_name,"is",net_price,"Dinars")
```

8. Program representation by flowcharts

Programs are designed to be run on computer. In reality, a program is stored as text in a disk-based file. Computers like to deal with text because it allows for simpler analysis and compilation. However, people are not always comfortable reading and comprehending vast volumes of text; they prefer visual representations (image, diagram, and so on).

Any algorithm can also be represented using a diagram known as a flowchart. Flowchart is a visual language in which the developer may express his algorithm in an understandable and straightforward manner. Obviously, such a representation is just designed to help the user understand the algorithm; it is not a programming language *per se*.

A flowchart is composed of boxes of different shapes to represent the steps of the algorithm. There are essentially four types of boxes:

- begin and end: which represent the beginning and the end of the algorithm. The box has a shape of a rounded rectangle. The box should contain the name of the step.
- Input and output actions: these actions are represented with a parallelogram. The box contains the instruction.
- Assignment actions: these actions are represented with a parallelogram. The box contains the instruction.
- Condition actions: these actions are represented with a diamond. They will be covered later.

Arrows are used to connect the boxes sequentially. If action a is followed by the action b, the we draw an arrow from the box of a to the box of b. The flowchart of the previous algorithm is the following: