

Conditional structures

Simple statements (assignments, input, output, pass) combined with expression cannot create much interesting programs. In fact, the linear flow of control is straightforward and does not allow a customization of processing according to the values of data. In particular, if we want to select data on which some processing might be required and not for other then we need conditional statements.

Programming languages have many kinds of conditional statements. In some kinds of them, the developer can explicitly specify the condition that should be satisfied to execute some other statements. In others, the developer can specify what happens when something goes wrong. In both case, the flow of control is no longer linear. The flowchart contains now branches (we talk about branching).

1. Simple conditional statement

A simple conditional statement is a statement that is composed of a logical expression and a statement. The latter can be an assignment, an input/output action, pass, a block, or even another conditional statement. The general form of this statement in the algorithm language is:

algo

```

1 if expression then
2 | statement

```

In Python, it is written as (a conditional statement always creates a new block):

py

```

1 if expression:
2 | statement

```

Note that the execution of the conditional statement is different according to the used language (in other words, we say that it has different *semantics* according to the language):

- In the algorithmic language: the expression is first evaluated. If the result equals true then statement is executed; otherwise, nothing happens (at least at this stage). The expression should evaluate to boolean, otherwise this is a compilation error.
- In Python: the expression is first evaluated. If the result equals True then statement is executed; otherwise, nothing happens. Many other values are equivalent to True in the conditional statement. Any non-null integer or float, any non-empty string, or any value different from None is also equivalent to True. Python, as a loosely typed language, is very permissive about that.

Example 3.1 : Absolute value

We wish to write a program that compute the absolute value of a variable. In the algorithmic language, we write:

```

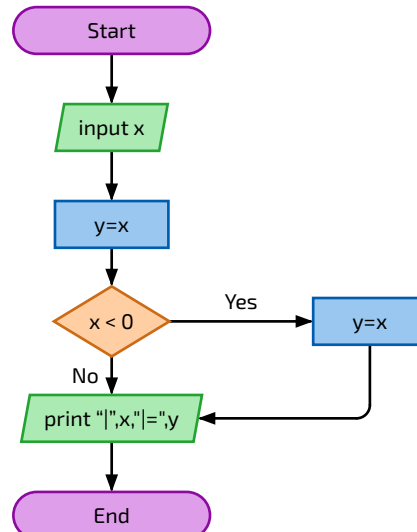
1 algorithm absolute
2 var x,y:float
3 begin
4   x=float(input("Give the value of x="))
5   y=x
6   if x < 0 then
7     y=-x
8   print("|",x,"|=",y)
9 end
  
```

In Python, we can write:

```

1 if __name__=="__main__":
2   x=float(input("Give the value of x="))
3   y=float(x)
4   if x < 0:
5     y=-x
6   print("|",x,"|=",y)
  
```

A conditional statement flowchart has a diamond with the condition written within. Two arrows emerge from the box: one labeled Yes represents the case in which the expression is evaluated as true. The other one is labeled No and represents the case in which the expression is evaluated as false. The first arrow connects the diamond to the internal statement of the conditional statement, whereas the latter connects it to the statement that follows the conditional statement. The flowchart in the above example will better illustrate this.



2. Compound conditional structure

The previous form of the `if` statement merely allowed us to indicate what to do if a condition was true. We often want to describe what to do if the condition is false. Cascade `if` statements are also commonly used when many conditions need to be tested successively.

To indicate what should be executed if the condition is false, we employ the full syntax of the conditional statement. In algorithmic language, this is done by:

algo

```

1 if expression then
2 |   statement_1
3 else
4 |   statement_2

```

while in Python, this is done by:

py

```

1 if expression:
2 |   statement_1
3 else:
4 |   statement_2

```

The execution of this compound conditional statement proceeds as follows: if the expression is evaluated to true (or any equivalent value to True in Python), then statement_1 is executed, otherwise statement_2 is executed.

It is also possible to have a cascade of tests in the algorithm language as follows:

algo

```

1 if expression_1 then
2 |   statement_1
3 else if expression_2 then
4 |   statement_2
5 ...
6 else
7 |   statement_n

```

In this case, expression_1 is first evaluated. If it is true, then statement_1 is executed, otherwise expression_2 is executed. If it is true, statement_3 is executed; otherwise, we continue with the remaining expressions in the same manner.

Python provides a specific syntax for cascading conditional statements. In fact, we use the 'elif' keyword, shown in the following code template (not using 'elif' in this case is a syntactic error):

py

```

1 if expression_1:
2 |   statement_1
3 elif expression_2:
4 |   statement_2
5 ...
6 else:
7 |   statement_n

```

Example 3.2 : Sign of a number

Consider a program that determines if an integer is positive, negative, or null. In the algorithmic language, we obtain:

algo

```

1 algorithm absolute
2 var x:float
3 begin
4 |   x=float(input("Give the value of x="))
5 |   if x > 0 then
6 | |   print(x,"is positive")
7 |   else if x < 0 then
8 | |   print(x,"is negative")
9 |   else
10 | |   print(x,"is null")
11 end

```

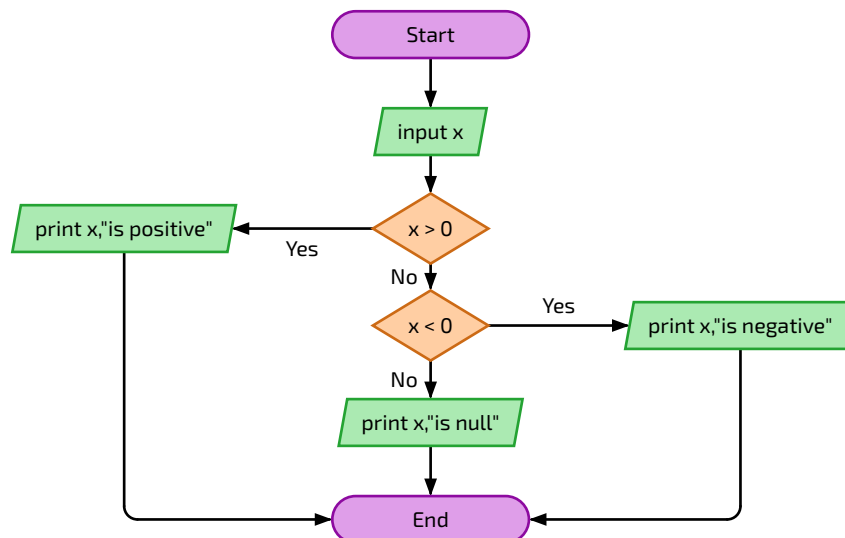
In Python, we get:

```

1 if __name__=="__main__":
2     x:float=float(input("Give the value of x="))
3     if x > 0:
4         print(x,"is positive")
5     elif x<0:
6         print(x,"is negative")
7     else:
8         print(x,"is null")

```

The flowchart diagram is similar to a simple conditional statement. We continue to employ a diamond with two branches. The flowchart for the preceding example is given by:



3. Conditional expression in Python

Python includes a very useful ternary operator that assigns a value to a variable based on a given condition. The general syntax is `v1 if expression_1 else v2`. This expression starts by evaluating `expression_1`; if it is `True`, then the whole expression evaluates to `'v_1'`; otherwise, its value is `v_2`. The Python version of the absolute value program can be rewritten as:

```

1 if __name__=="__main__":
2     x:float=float(input("Give the value of x="))
3     y:float=x if x>0 else -x
4     print("|",x,"|=",y)

```

4. Multiple choice conditional structure

When there are too many conditions in a cascade `If` statement, the program becomes less readable. In the algorithmic language, an alternative way to test the value of an expression and do different stuff for each value can be done by the syntax `switch...case...end`. The general form is given by:

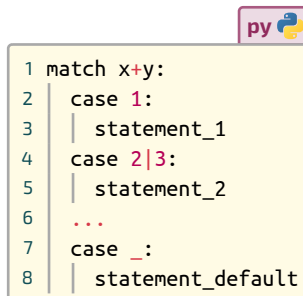
```

1 switch expression
2 begin
3     case v_1:statement_1
4     case v_2:statement_2
5     ... default:statement_else
6 end

```

The switch statement can be seen (in the algorithmic language) as a *syntactic sugar* of a cascade of if statements. The execution of these statements starts by evaluating the expression. Then its value is compared to `v_1`; if they are equal, then `statement_1` is executed. Otherwise, the same steps are repeated for each value. If no value matches the value of `expression`, then the default statement `statement_else` is executed. The flowchart of this statement is similar to a compound condition statement.

Python lacked a multiple-choice conditional structure until version 3.10. This was achieved by simply employing a cascade of if statements. Version 3.10 introduced a new multiple-choice statement, `match...case`. As the name implies, this is a pattern matching statement rather than a multiple-choice statement. A full description of this statement is beyond the scope of this course; we will just look at the most basic use case. Let's take an example.



```

1 match x+y:
2     case 1:
3         statement_1
4     case 2|3:
5         statement_2
6     ...
7     case _:
8         statement_default

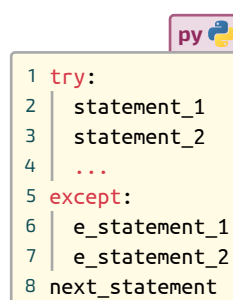
```

As we can see here, the `match` syntax is richer. In this example, the expression `x+y` is first evaluated. If its value is 1, then `statement_1` is executed. If its value is 2 or 3, `statement_3` is executed. If no value corresponds to the value of the expression, (case `_`), then `statement_default` is executed.

5. Handling errors

Programming languages provide a range of error handling mechanisms. For example, dividing by zero results in an error, as does transforming a non-number string into integer. In Python, errors are handled by raising and catching *exceptions*. When a faulty code is executed (for example, a division by zero), an exception is raised, and the current *function* is halted. If no code is intended to handle the error, the whole program is terminated, and an error message is displayed. Python offers a way for running code when an exception is raised. This is done using the syntax 'try...except'. Despite the fact that exception handling is rather an advanced feature in Python, we will look at some easy examples for dealing with errors.

Basically, error handling is comparable to branching. Python tries to run a code; if it fails, another code is executed. The general (and simplest) syntax for this is:



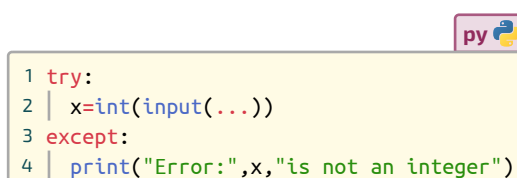
```

1 try:
2     statement_1
3     statement_2
4     ...
5 except:
6     e_statement_1
7     e_statement_2
8 next_statement

```

This code is executed as follows. Python starts by executing the block `statement_1 statement_2...`. If no error occurs, then it skips the block of `except` and proceeds to execute `next_statement`. Otherwise, the execution of the first block is halted, and Python starts executing `e_statement_1 e_statement_2...`, followed by `next_statement`.

For instance, this code can be used to verify if a string corresponds to an integer:



```

1 try:
2     x=int(input(...))
3 except:
4     print("Error:",x,"is not an integer")

```

We should note that this code is not quite proper since it does not define what type of error it handles, but we will not go further into this topic for the moment.