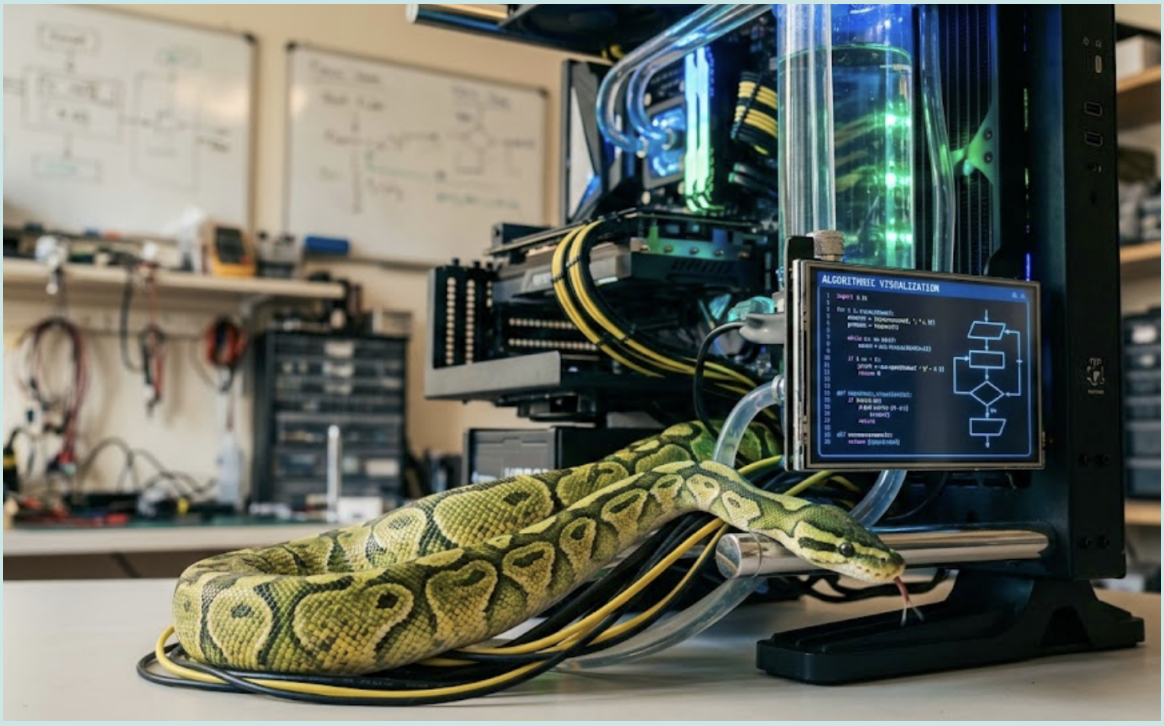


Initiation à l'algorithmique

(et au langage Python)

Jean-Marc DESBONNEZ



– 2026 –

1. Généralités

1.1. Quoi ?

Le mot *algorithme* est d'origine arabeă; le mathématicien *Muhammad Ibn Musa al Kharzami* (début du IX^{e} siècle) utilisait des ensembles de symboles et de procédés de calculs mathématiques.

A l'origine, un algorithme est donc un ensemble de règles opératoires propres à un calcul. Certains algorithmes sont célèbres : calcul d'une racine carrée (Héron D'ALEXANDRIE), calcul de décimales du nombre π, algorithme d'EUCLIDE (300 av. J.C.) pour le calcul du P.G.C.D. (1) de deux nombres, etc.
Ces règles opératoires consistent à décomposer un calcul complexe en un enchaînement de calculs élémentaires.

De tels procédés de calculs sont utilisés dans les calculettes et les ordinateurs, c'est à dire dans des machines capables uniquement de faire des calculs élémentaires en langage binaire. À l'ère de l'informatique, on fait faire par un ordinateur presque n'importe quoi ; il « suffit » d'écrire l'enchaînement des actions nécessaires à l'accomplissement d'une tâche donnée, l'ordinateur jouant le rôle de *robot bête et discipliné* (mais rapide...).

> L'algorithmique consiste à décomposer un problème complexe en une succession de tâches élémentaires.

C'est « l'art du saucissonage ».

1.2. Exemple

Dans la vie de tous les jours, nous côtoyons beaucoup d'algorithmes : n'importe quel « mode d'emploi », de la machine à lessiver au lecteur DVD en passant par le GSM, est finalement un algorithme, puisqu'il consiste en une succession d'instructions simples que l'utilisateur final est sensé devoir (savoir) accomplir.

Détaillons par exemple une banale promenade à vélo :

un humain ne va (heureusement) pas faire ce genre de raisonnement, et tous ces gestes seront opérés de manière naturelle, mais s'il faut mettre un robot sur le vélo, il faudra bien *lui dire* ce qu'il doit faire !

(1). Plus Grand Commun Diviseur

```
Début
Mettre une tenue adéquate
Prendre le vélo
Si problèmes techniques
      Alors réparer
Monter sur le vélo
Tant que non arrivé à destination
      Pédaler
      Si montée ou descente
            Alors changer de vitesse
      Se reposer un peu
/
/
Remonter sur le vélo
Tant que non arrivé à destination
      Pédaler
      Si montée ou descente
            Alors changer de vitesse
Prendre une douche
Fin
```

Il est même toujours possible de détailler certaines actions de manière encore plus fineă:

```
Si problèmes techniques
      Selon que
            Pneu dégonflé
                  Regonfler
            Selle trop haute
                  Baisser selle
            Selle trop basse
                  Lever selle
            Chaîne rouillée
                  Graisser chaîne
            Etc.
```

Et pour pousser le bouchon un peu plus loin encoreă:

```
Selle trop haute :
      Prendre clé
      Desserrer écrou
      Tant que pas à la hauteur
            Baisser
      Serrer écrou
      Ranger clé
```

A ce rythme-là, on constate qu'il n'y a plus de fină!
L'important est de savoir à quel niveau de détails il faut s'arrêter. C'est ce que l'on appelle les actions de base, ou, dans le jargon informatique, les « primitives du langage ».

1.3. Informatiquement parlant

En informatique, l'algorithmique est l'étape préliminaire (et indispensable) à la réalisation deă*programmes*, aussi appelés *logiciels* ou encore *macro-instructions* dans certains langages de programmation.

Un programme est une succession d'instructions élémentaires écrites dans un langage qui sera interprété et exécuté par la machine.
Avant la phase de traduction dans un langage de programmation, il est nécessaire de réaliser un « plan » dans lequel on tiendra compte de toutes les éventualités qui pourraient se présenter, des erreurs possibles dues à l'utilisateur, etc.
Faire ce plan, c'est écrire l'algorithme du problème. C'est la phase la plus complexe, car il faut penser à toută; la phase de traduction n'est que technique, il suffit d'appliquer quelques *règles de grammaire*, la *syntaxe* du langage de programmation.

L'algorithme doit être, en principe, **indépendant** de tout langage de programmation.

On peut comparer la réalisation d'un programme à la construction d'une maisonă: il faut d'abord que l'architecte fasse les plans, et ensuite la phase de construction par l'entrepreneur

peut commencer.
En principe, les plans doivent être compréhensibles par n'importe quel entrepreneur, de n'importe quelle région, pour autant évidemment que le terrain soit conforme.
Dans l'écriture des algorithmes, on utilise fréquemment des TESTS, encore appelés CONDITIONSă; ce sont des expressions qui sont soit vraies, soit fausses, et qui orientent la suite des opérations.
C'est l'*algèbre de Boole* qui régit les lois concernant ces expressions, aussi appelées *expressions booléennes*, du nom du mathématicien (2) qui s'y appliqua.

1.4. Schéma de travail

Ce schéma de travail est INDISPENSABLE à la création d'un « logiciel », quel qu'il soit.
Au départ, un problème est posé, par exemple « calculer la somme des 50 premiers nombres pairs ».
À partir du problème, il faut en écrire l'algorithme, c'est-à-dire la succession des actions élémentaires qui vont aboutir à la solution du problème. Cette succession d'actions élémentaires est écrite dans un *langage structuré*, appelé *pseudo-code*, proche des langages informatiques modernes, tout en restant le plus indépendant possible d'un langage particulier.
L'étape suivante, technique, est la traduction de l'algorithme dans un langage informatique approprié au problème de départ.
Le résultat de la traduction de l'algorithme dans un langage de programmation est appelé code sourceă; il est réalisé par l'intermédiaire d'un éditeur de textes spécialisé.
Ce code source sera ensuite *transformé* en code exécutable via un compilateur associé au langage de programmation. C'est ce code exécutable qui est appelé *programme*, *logiciel*, ou *macro instruction*, ...

On verra par la suite que le langage *pseudo-code* est un langage très simple, avec peu d'instructions et de structures, et donc assez simple à mettre en oeuvre.
Il est calqué sur la plupart des langages de programmation modernes, ce qui le rend très « universel ».

Au début de l'informatique, un algorithme consistait en un plan (3) composé de figures géométriques ayant chacune une signification bien précise, et reliées par des flèches indiquant la suite des calculs à effectuer. Cette manière de voir ne convient plus aux langages actuels.

1.5. Un exemple concret

Pour donner au lecteur l'envie de lire la suite, voici **un** (on pourrait en imaginer d'autres) algorithme qui permet de calculer la somme des 50 premiers nombres pairs.

(2). George Boole (1815–1864).
(3). Appelé *ordinogramme*.

1.5.1. Pseudo-code et traduction en Python

```
Pseudo-code

somme ← 0
compteur ← 1
tant que compteur < 50
        somme ← somme +2*compteur
        compteur ← compteur + 1
fin du tant
écrire somme
```

Traduction en langage Python :

```python
# sompairs.py
# somme des 50 premiers nombres pairs

somme = 0
compteur = 1
while compteur < 50 :
  somme = somme + 2*compteur
  compteur = compteur + 1
print("la somme vaut",somme)
```

Et la réponse est : 2450 ☺

1.5.2. Un peu d'explications

 # : c'est le caractère qui précède une ligne de commentaire, ligne qui sera ignorée par le compilateur ou l'interpréteur. Il est fondamental de commenter des lignes de code par des explications, pour se souvenir !

somme, compteur : ce sont des *variables*, espaces mémoire identifiés par un nom.

← opération d'*affectation*, qui consiste à mettre un contenu dans une variable ; dans beaucoup de langages de programmation, elle est traduite par le signe « = ».

somme ← somme+2*compteur : on prend le contenu de la variable *somme*, on y ajoute le double de la variable *compteur* ; le résultat est ré-affecté à la variable *somme*.

compteur ← compteur+1 : idem que ci-dessus : on prend le contenu de la variable *compteur*, on y ajoute 1 ; le résultat est ré-affecté à la variable *compteur*.
Cette opération est appelée *incrémentation* de la variable *compteur*.

tant que : c'est une des structures de type *répétitive* ; toutes les instructions de la structure seront exécutées tant que la condition d'arrêt n'est pas vérifiée ; logiquement traduite par « *while* ».

écrire somme : consiste à afficher (à l'écran) le contenu de la variable *somme* ; traduction en python par l'instruction « print() ».

2. Python, un langage de programmation

Faire de l'algorithmique sans utiliser un langage de programmation c'est un peu faire du solfège sans jouer d'un instrument : c'est frustrant !
Rien de tel en effet pour savoir si l'algorithme fonctionne bien que de « faire jouer la partition » par l'ordinateur lui-même.

On sera donc amené à courir deux lièvres simultanément :

- ➙ L'algorithmique.
- ➙ Le langage de programmation.

2.1. Langage de programmation

2.1.1. Rôle

Traduction logique : un langage de programmation est un langage dans lequel on écrit des programmes (ensembles d'instructions exécutables par un ordinateur).

Au départ (et à l'arrivée aussi d'ailleurs), le seul langage exécutable par un ordinateur est le *langage binaire* : langue extrêmement basique, puisque composée de 2 « mots » uniquement, 0 et 1.
Pour un humain, il est assez indigeste ; à essayer de retaper sans se tromper :

10100100111010110010101001100101011101101011011101010010010101 0

ou d'écrire aisément le mot « MATH » sachant que le code binaire du A est 01000001 (4), celui du H est 01001000, celui du M est 01001101 et celui du T est 01010100.

Pour surmonter cette difficulté, les informaticiens ont mis au point ce qu'on appelle des *langages de haut niveau*.

Ce sont des langages techniques, dont le vocabulaire est relativement « humain », avec des mots à consonance anglaise (universalité oblige).

De manière imagée, on peut dire qu'ils fonctionnent comme un dictionnaire *humain-binaire*, à savoir qu'ils traduisent en binaire des instructions écrites en « humain » afin de permettre leur exécution par une machine qui ne comprend que le binaire.

Le codage en binaire est appelé *compilation*.

2.1.2. Lequel choisir ?

Il existe de (très) nombreux langages de programmation ; ils ont bien sûr beaucoup évolué, en même temps que l'informatique ; ils ont parfois un domaine d'application particulier : plutôt internet, plutôt scientifique, plutôt gestion, ..., et sont aussi sujets au phénomène de mode.

(4). Code ASCII, American Standart Code for Information Interchange.

Certains sont aussi connus que les dialectes du fond de l'Amazonie, d'autres font plus parler d'eux .

Pour ne citer (chronologiquement) que ceux auxquels je me suis frotté : FORTRAN, ASSEMBLEUR, BASIC, TURBO BASIC, PASCAL, CLIPPER, VISUAL BASIC, APPLESCRIPT, C, JAVA, RUBY, et enfin PYTHON.

Certains sont plus faciles à apprendre que d'autres — le pire, c'est C — (l'*anglais* est plus facile à apprendre que le *chinois*), mais dans tous les cas, c'est 1 heure de théorie pour 24 heures de pratique... c'est un métier, mais surtout une passion.

2.1.3. Python !

Pourquoi Python ?

Python est l'un des langages de programmation les plus populaires. Il a été créé en 1991 par Guido van Rossum (1956 –), Pays-Bas.
Il est réputé pour sa simplicité et sa polyvalence. Sa syntaxe est claire et lisible. Le code ressemble à de l'anglais et utilise l'**indentation** (5) pour structurer le code, ce qui le rend facile à lire.
Il est multi-usage et très puissant : IA, développement web, calculs scientifiques, ...
Il existe des centaines de bibliothèques d'instructions prêtes à l'emploi, et est utilisé par des géants de la tech (Google, Netflix, NASA, ...)
Et enfin, il est gratuit et porté par une immense communauté. C'est donc un langage idéal pour notre quête d'**initiation** à l'algorithmique.
Nous ne chercherons pas à optimiser les lignes de code, laissons cela aux pros !
Le lecteur qui souhaitera progresser trouvera suffisamment de documentation dans la littérature spécialisée, sur le net, et aussi chez le « chat péteur », comme dirait mon ami Pascal, l'IA étant assez efficace sur des questions en rapport avec les langages de programmation.

2.1.4. Installation et mise en place

Je ne suis plus « windosien » depuis de nombreuses années, je ne peux décrire que l'installation sous MacOS, qui est d'ailleurs très simpliste, comme souvent.

1. sur le site python.org/downloads, télécharger la version correspondant à votre modèle de machine ; c'est un fichier de type .pkg.
2. exécuter ce fichier... et c'est terminé.
3. on peut vérifier via le *terminal* : `python3 -version`, un message indiquera la version installée.

```
imac:~ jm$ python3 --version
Python 3.14.3
```

2.2. Python interactif

Pour débuter, on peut l'utiliser Python en *mode interactif*, à savoir que, dès qu'une instruction est encodée, elle est exécutée.

(5). Décalage des lignes.

Via le terminal, taper l'instruction `python3`.

```
imac:~ jm$ python3
Python 3.14.3 (v3.14.3:323c59a5e34, Feb  3 2026, 11:41:37) [Clang 16.0.0 (clang-1600.0.26.6)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello world !")
hello world !
>>> 3.13159 / 5
0.626318
>>> exit()
imac:~ jm$
```

Le message d'invite devient alors `>>` et il suffit de taper une instruction ; elle est immédiatement exécutée par l'interpréter (si toutefois la syntaxe est correcte...)

Bien évidemment, ce mode a des limites, et après avoir pris en main les instructions de base, il faudra passer en *mode éditeur* (on tape l'ensemble des instructions dans un fichier dont l'extension **doit** être `.py`, puis on exécute ce fichier.

2.2.1. Les opérations arithmétiques de base

Il y en a 7 : addition(+), soustraction (-), multiplication (*), division (/), division entière (//), exponentiation (**) et modulo (%).
A essayer successivement...
$8 + 9.41 \rightsquigarrow$ (le séparateur décimal est le point)
$8 - 9$
$8 * 9$
$14/11 \rightsquigarrow 1.2727272727272727$
$8 * *9 \rightsquigarrow 134217728$
$8 * *9 * *4 \rightsquigarrow$ désolé, pas assez de place
$432\%7 \rightsquigarrow 5 \quad \left(\frac{432}{7} = 61 \text{ reste } 5\right)$
$432//7 \rightsquigarrow 61$

Bien évidemment, la priorité des opérations est toujours respectée, mais on peut modifier ces priorités en utilisant des parenthèses.

Ces opérations arithmétiques de base ne nécessitent pas l'importation d'une bibliothèque particulière, elles sont natives à Python.

2.2.2. Les calculs mathématiques un peu plus poussés

Ces opération nécessitent l'importation de la bibliothèque *math* à l'aide de l'instruction

```
import math
```

Chaque fonction doit alors être précédée du préfixe `math.`

Pour calculer $\sqrt[n]{x}$, avec $n > 2$, on utilise `x ** (1/n)`.

Fonction	Explication
`math.sqrt(x)`	$\sqrt{x}$
`math.sin(x)`	sinus de x en radian
`math.cos(x)`	cosinus de x en radian
`math.tan(x)`	tangente de x en radian
`math.exp(x)`	e^x
`math.log(x)`	logarithme naturel (ou népérien) de x
`math.log10(x)`	logarithme vulgaire (ou décimal) de x
`math.log(x,a)`	logarithme en base a de x ($\log_a x$)
`math.e`	le nombre d'Euler e=2.718281828459045
`math.pi`	π = 3.141592653589793

TABLE 1 : Opérateurs mathématiques de base

2.2.3. Mode interactif et algorithme

En mode interactif, on peut taper *successivement* les instructions d'un algorithme : taper par exemple celles permettant de calculer la somme des 50 premiers nombres pairs (voir instructions page 4).

L'encodage des instructions « ne produit rien », sauf la dernière, qui affiche le résultat.

Inconvénients majeurs de cette manière de procéder :

- Si on a fait une faute de frappe en cours de route, on le verra (peut-être) au résultat final, et il faut tout recommencer !
- Si on souhaite calculer la somme des 60 premiers nombres pairs, il faudra tout retaper, alors qu'il ne suffit que d'une petite modification à la ligne 6.
- Et pour des algorithmes de 4000 lignes ?

On voit ici les limites du mode interactif.

2.3. Python en mode éditeur

2.3.1. Éditeur de textes

Le mode *éditeur* consiste à taper toutes les instructions de l'algorithme dans un document de type *texte*, à l'aide d'un logiciel appelé « éditeur de textes ». Les instructions ne sont pas interprétées (et donc pas exécutées) par Python au fur et à mesure de leur frappe ; c'est comme si on écrirait une « liste de courses ».

Ce document, appelé ***code source*** doit être enregistré sur le disque dur (attention, il y a des conventions pour le nom [6]) et pourra être exécuté par la suite au moyen d'une instruction via le Terminal.

Choix d'un éditeur

Il ne faut pas confondre « éditeur de textes » avec « traitement de textes ». Un éditeur de texte ne possède pas d'outils sophistiqués de mises en forme, de gestion de tableaux, de création de tables des matières, etc.

Pour taper des instructions, le minimum nécessaire est la numérotation et l'indentation des lignes (souvent activées par défaut) ; un plus bien appréciable est la coloration syntaxique des mots-clé du langage utilisé.

On trouve de nombreux éditeurs de textes gratuits sur internet ; chaque système d'exploitation en propose d'ailleurs un par défaut : *Textedit* sur MacOsX, *Wordpad* sur Windows.

Ils sont suffisants pour Python, mais ne permettent pas de numéroter les lignes (important pour de longs algorithmes) ni la coloration syntaxique du langage. De plus, pour exécuter les fichiers, il faut passer par le terminal (MacOS) ou WindowsTerminal (Windows).

Il vaut peut-être mieux opter pour des logiciels spécialisés comme Visual StudioCode ou Spider (gratuits) ou BBEdit (maintenant payant). Le gros avantage est qu'ils possèdent un « bouton » permettant d'exécuter le script ouvert.

(6). Doit commencer par une lettre, ne contenir que des lettres et des chiffres, pas d'espace, pas de lettres accentuées, pas de signes de ponctuation, l'extension doit être **.py**

3. Algorithmique et Python

Tout comme un maçon n'entreprend pas la construction d'une maison sans un plan de l'architecte, on n'entreprend pas la création d'un programme sans l'algorithme correspondant.

Dans ce chapitre, on mettra en place un *langage structuré* facilement transposable en un langage de programmation.

3.1. Variable et constante

Pour qu'une information soit traitée par un ordinateur, il faut qu'elle y soit mémorisée.

Une *variable* est un espace mémoire étiqueté (le nom de la variable) dont le contenu peut varier au cours des instructions.
Une *constante* est également un espace mémoire étiqueté, mais dont le contenu restera fixe (interdiction et impossibilité de modifier le contenu).

Les deux peuvent contenir entre autre un nombre, un texte (appelé chaîne), etc. Dans le cas d'une variable, un ancien contenu sera remplacé par un nouveau (le suivant écrase le précédent).

Il y a plusieurs types de variables :

- la variable **simple**, que l'on peut comparer à un tiroir ;
- la variable **tableau**, que l'on peut comparer à une commode de plusieurs tiroirs ;
- la variable **booléenne**, qui contient soit *true*, soit *false*, très utiles pour les structures alternatives.

3.2. Nom d'une variable et d'une constante (règles Python)

Ces règles peuvent toutefois changer d'un langage à l'autre.

Un nom de variable ne peut pas commencer par un chiffre, mais par une lettre (ou le symbole *underscore*), peut contenir des lettres, des chiffres, le, symbole *underscore*, mais pas de caractère accentué, ni d'espace ni de signes de ponctuation.

Python est sensible à la casse (distingue majuscule et minuscule) : `Prix` et `prix` sont deux variables distinctes. Une bonne pratique est de mettre les constantes en majuscules : `TAUX_TVA=0.25` (le contenu d'une constante, comme son nom l'indique, ne varie pas au cours de l'exécution du script).

Et, surtout, donner des noms explicites : `resultat`, `date_naiss`, `factorielle` plutôt que `r`, `d`, `f`.

3.3. Le langage structuré, en résumé

C'est un langage technique, simple, avec peu de syntaxe.

Il est composé de

— 3 instructions de base : l'affectation, la lecture et l'écriture (certains langages comme *Java* par exemple en nécessitent une quatrième, la déclaration).

— 3 structures : la séquence, l'alternative, la répétitive.

3.4. La séquence

Une séquence d'instructions n'est rien d'autre qu'une suite d'instructions ; elles sont exécutées dans l'ordre dans lequel elles sont écrites (d'où la numérotation des lignes dans l'éditeur).

3.5. L'affectation

L'affectation consiste à mettre un contenu dans une variable ; le contenu peut être une donnée fixe, le résultat d'un calcul, le contenu d'une autre variable, etc.

langage structuré	Python
variable ← contenu	variable = contenu
prenom ← "jules"	prenom = "jules"
prix ← 20	prix = 20
taux ← 0.3	taux = 0.3
reduction ← taux * prix	reduction = taux * prix
prix ← prix − reduction	prix = prix − reduction

Avec l'*affectation*, c'est la personne qui écrit le programme qui « décide » du contenu de la variable (ne pas confondre avec la *lecture* où c'est l'utilisateur final qui décidera).

3.6. La lecture

La lecture est une instruction dite d'*entrée* ; elle permet de faire « entrer » une donnée de l'extérieur (l'utilisateur final, via le clavier) vers l'intérieur (une variable).

Il s'agit d'une « affectation de l'extérieur ».

Attention : si on demande à l'utilisateur d'encoder son âge et qu'il encode `vingt-cinq`, c'est mort pour les calculs éventuels qui suivent… une instruction de lecture doit (toujours) être suivie d'un test d'erreur, qui est souvent complexe, et sort du champ de l'initiation. Nous lui préférerons donc une simple affectation.

3.7. L'écriture

L'écriture est une instruction dite de *sortie* ; elle permet de faire « sortir » un résultat vers l'extérieur, l'écran en ce qui nous concerne.

langage structuré	Python
écrire donnée	print()

La syntaxe de l'instruction `print()` dépend du type de donnée à écrire (chaîne ou nombre). Dans le cadre de notre initiation, les 3 exemples qui suivent devraient suffire.

instruction	effet
print (prix)	affiche le contenu de la variable *prix*.
print ("blabla")	affiche le texte "blabla".
print ("blabla", prix)	affiche le texte "blabla" suivi du contenu de la variable *prix*.

3.8. Quelques exercices, pour s'échauffer

1. On encode 2 nombres ; calculer (et afficher, évidemment) leur somme, leur produit et la différence entre le premier et le second.
2. On encode les 3 coefficients d'une équation du second degré (celui de x^2 sera supposé non nul). Calculer le discriminant de l'équation.
3. On encode le rayon d'un cercle. Calculer sa circonférence et sa surface.
4. On encode la longueur des 2 côtés de l'angle droit d'un triangle rectangle. Calculer la longueur de son hypoténuse.
5. Permuter le contenu de 2 variables.
6. À partir du rayon d'un cercle, calculer sa longueur et son aire.

3.9. La structure alternative

C'est une structure qui permet de « prendre une décision » en fonction d'une condition soit vraie soit fausse.

Première forme, alternative simple (1 ou 2 possibilités)

```
si condition
   instruction(s) cas vrai
sinon
   instruction(s) cas faux
fin
```

```
if condition :
   instruction(s) cas vrai
else :
   instruction(s) cas faux
```

Exemple

```
lire nombre
si nombre est pair
   écrire "nombre pair"
sinon
   écrire "nombre impair"
fin
```

```
1 nombre = 129
2 if nombre % 2 == 0 :
3   print(nombre, "est pair")
4 else :
5   print(nombre, "est impair")
```

Remarque

Le cas *sinon* peut ne pas exister !

Exemple

```
lire nombre
si nombre est premier
   écrire "nombre premier"
fin
écrire "suite..."
```

```
1 import sympy as sp
2 nombre = 1000033
3 if sp.isprime(nombre) :
4   print(nombre, "est premier")
5 print("suite...")
```

La bibliothèque *sympy* est incontournable pour faire du calcul symbolique (développer et factoriser une expression, résoudre une équation algébrique, calculer une dérivée par rapport à x, calculer une intégrale définie ou indéfinie, résoudre une équation différentielle [7], etc.) ainsi que le test de primalité qui nous intéresse ici.

Il est de tradition de donner un nom raccourci à une bibliothèque importée, le nom de la fonction utilisée doit alors être précédée de ce nom raccourci suivi d'un point.

(7). On trouve sur le net, ou avec l'aide de l'IA, des exemples d'utilisation.

Seconde forme, alternatives imbriquées (plus de 2 possibilités)

```
si cond1
   instruction(s)1
sinon si cond2
   instruction(s)2
sinon si cond3
   instruction(s)3
...
sinon
   autre instruction
fin
```

```
if cond1 :
   instruction(s)1
elsif cond2 :
   instruction(s)2
elsif cond3 :
   instruction(s)3
...
else :
   autre instruction
end
```

Exemple classique : résolution d'un second degré

```
lire a, b, c
delta <-- b*b-4*a*c
si delta > 0 :
   écrire "2 solutions réelles distinctes"
   x1 <-- (-b-racine(delta))/(2*a)
   x2 <-- (-b+racine(delta))/(2*a)
   écrire x1, x2
sinon si delta == 0 :
   écrire "1 solution réelle"
   x <-- -b/(2*a)
sinon :
   écrire "pas de solution réelle"
fin
```

```python
import math
a, b, c = 1, 7, -2
delta = b**2 - 4*a*c
if delta > 0 :
  print("2 solutions reelles distinctes")
  x1 = (-b-math.sqrt(delta))/(2*a)
  x2 = (-b+math.sqrt(delta))/(2*a)
  print(x1, "et", x2)
elif delta == 0 :
  print("1 solution reelle")
  x = -b / (2*a)
  print(x)
else :
  print("pas de solution reelle")
```

Expressions booléennes

Une *expression booléenne*, encore appelée *condition*, est une expression dont la valeur est soit vrai (true) soit faux (false).

On les retrouve dans les structures alternatives, ainsi que dans les structures répétitives.

Expression	Explication
a == b	a est identique à b
a < b	a est strictement inférieur à b
a <= b	a est inférieur à b
a > b	a est strictement supérieur à b
a >= b	a est supérieur à b
a ! = b	a est différent de b
(cond1) or (cond2)	cond1 ou cond2 (est fausse si les 2 sont fausses)
(cond1) and (cond2)	cond1 et cond2 (est vraie si les 2 sont vraies)
not (cond1)	négation de cond1 (est vraie si cond1 est fausse, et vice-versa)

TABLE 2 : Opérateurs de comparaison et expressions booléennes

3.10. Exercices

1. Pour un nombre donné,
   — calculer sa valeur absolue (il faudra se rappeler de la définition de valeur absolue...) ;
   — calculer son inverse (attention, zéro n'a pas d'inverse...) ;
   — calculer sa racine carrée (attention, un négatif n'a pas de racine carrée...).
2. Ecrire un algorithme qui valide une heure (exprimée en heures, minutes et secondes) à partir de 3 entiers donnés.
3. Simuler le lancer d'un dé, et afficher le résultat sous la forme "un", "deux", ... "six".
4. À partir d'un résultat d'une interrogation (supposé compris entre 0 et 20), calculer (et afficher) une appréciation selon la grille suivante :

Résultat	Appréciation
$\in [0,5[$	nettement insuffisant
$\in [5,9[$	insuffisant
$\in [9,10[$	faible
$\in [10,13[$	satisfaisant
$\in [13,15[$	bien
$\in [15,20]$	très bien

5. Afficher 2 nombres en ordre croissant.
6. Afficher 3 nombres en ordre croissant.
7. Vérifier si une année a est bissextile (elle doit être divisible par 4 ou par 400, mais pas par 100).

3.11. La structure répétitive, dite « boucle »

Une répétitive est une structure, qui comme son nom l'indique, permet d'exécuter plusieurs fois une ou plusieurs instructions, l'arrêt étant déterminé par une *expression booléenne*.

La règle fondamentale :

une répétitive DOIT s'arrêter !

Il est donc indispensable que parmi les instructions qui seront répétées, une au moins permette à l'expression booléenne de devenir soit *vraie* soit *fausse*, selon le type de répétitive.

Tous les langages de programmation proposent 2 types de répétitives :
- la répétitive *automatique* : le programmeur connaît à l'avance le nombre d'itérations.
- la répétitive *non automatique* : le nombre d'itérations dépend d'une condition.

3.11.1. La répétitive automatique (for)

```
Pour compteur de debut à fin
   instruction(s)
fin
```

```
n = 10
fact = 1
for k in range(1, n+1) :
  fact = fact * k
print (fact)
```

La variable *compteur* (k) est initialisée à la valeur *debut* (1) ; après chaque itération elle est incrémentée de 1 (par défaut) ; la répétitive s'arrête dès que la variable *compteur* a atteint la valeur *fin* exclue.

debut et *fin* doivent être des entiers, et peuvent être négatifs.

Et bien évidemment, pour que la répétitive s'arrête, *debut* $\leqslant$ *fin* !

Le petit script ci-dessus calcule le nombre 10!.

Si on veut que l'incrémentation soit différente de 1, le pas est le troisième argument de la fonction `range` : for k in range (1, 100, 2).

3.11.2. La répétitive non automatique (while)

Dans ce cas, le programmeur ne connaît pas à l'avance le nombre d'itérations qui seront effectuées. Attention, dans les instructions répétées, il faut **absolument** une instruction qui permet à la condition d'arrêt d'être vérifiée !

```
Tant que condition
   instruction(s)
fin
```

```
k = 3
while k<=12 :
  print(k)
  k = k+1
```

3.12. Exercices

1. Ecrire un algorithme qui permet de répondre à la question suivante :
« si on additionne les nombres entiers consécutifs $1+2+3+\cdots$, à partir de quel entier la somme dépassera-t-elle 10000 ? »

2. Compter le nombre de lancers de 2 dés avant d'obtenir un double six.
Python possède une instruction qui permet d'engendrer un nombre *pseudo aléatoire* : `randint()` ; elle nécessite la bibliothèque `random` [8].
```
import random
lancer = random.randint(1,6)
```

3. Calculer n termes de la suite de Fibonacci : $x_0 = 1$, $x_1 = 1$, $x_n = x_{n-1} + x_{n-2}$.
Calculer aussi les rapports $\frac{x_n}{x_{n-1}}\ldots$ bonjour le nombre d'or.

4. Calculer n termes de la suite de Padovan : $x_0 = 1$, $x_1 = 1$, $x_2 = 1$, $x_n = x_{n-2} + x_{n-3}$.
Calculer aussi les rapports $\frac{x_n}{x_{n-1}}\ldots$ bonjour le nombre de plastique.

5. La suite de Syracuse (ou conjecture de Collatz [9]) : à partir d'un nombre naturel n, on le divise par 2 s'il est pair, on le multiplie par 3 et on ajoute 1 s'il est impair ; une fois qu'on atteint 1, la suite entre dans une boucle infinie.
Calculer
   - le temps de vol : le nombre d'étapes nécessaires pour atteindre 1 ;
   - l'altitude : la plus grande valeur atteinte par la suite ;
   - le temps de vol en altitude : le nombre d'étapes pendant lesquelles la suite reste au-dessus de sa valeur initiale.

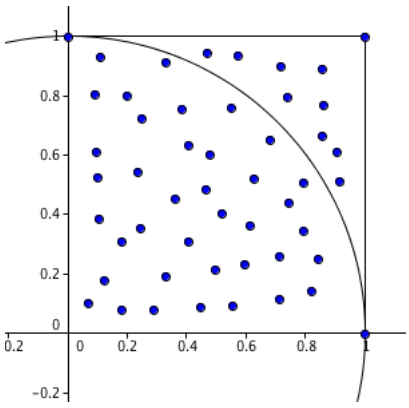
6. Vérifier si un nombre n est premier (seulement 2 diviseurs) sans utiliser la fonction `isprime()`.

7. A partir d'un naturel n, calculer
   (a) la somme $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n}$
   (b) la somme $\frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \ldots + \frac{1}{n!}$

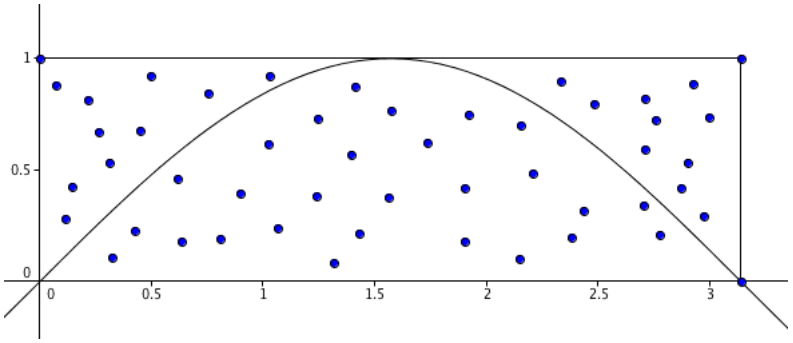
   comparer la valeur obtenue avec e le nombre d'Euler.

8. Calcul approché de l'aire d'un cercle de rayon 1 (et donc aussi d'une valeur approchée de π) par la méthode de *Monte Carlo*. On imagine le premier quart du cercle trigonométrique inscrit dans un carré de côté 1.
On "bombarde" ce carré d'un certain nombre n de points *aléatoires* dont l'abscisse et l'ordonnée sont dans l'intervalle $[0,1]$ et on compte le nombre x de points situés dans le quart de cercle.
Le rapport $\frac{x}{n} \cong \frac{\text{aire du quart de cercle}}{\text{aire du carré}}$ d'où l'on tire assez aisément une (mauvaise) valeur approchée de π.

[8]. À cette bibliothèque on ne donne pas de nom raccourci, car elle fait partie de la bibliothèque standard de Python (tout comme la bibliothèque *math*).
[9]. Peu importe le nombre de départ, la suite finit toujours par atteindre 1 (non encore démontré, vérifié jusqu'à $n = 2^{68}$).



On peut également utiliser cette méthode pour approcher l'aire d'une surface plane délimitée par le graphique d'une fonction quelconque et l'axe des abscisses (calcul d'une *intégrale définie*).
Par exemple une valeur approchée de l'aire de la surface plane délimitée par le graphique de $\sin(x)$ et l'intervalle $\left[0, \pi\right]$.



9. Le *jeu de crap*
On lance 2 dés et on fait la somme S des points obtenus.
Si S=7 ou 11, c'est gagné ; si S=2, 3 ou 7 c'est perdu.
Dans les autres cas, on relance les dés jusqu'à ce que la somme P des résultats vaut 7, et c'est perdu, ou S et c'est gagné.

10. Lancer 2 dés jusqu'à obtenir un doublé ; calculer le nombre de jets nécessaires.

11. Lancer 2 dés n fois et calculer les fréquences des sommes obtenues.

4. Si la sauce a bien pris...

La suite logique, une fois l'étape d'initiation assimilée, on cherche à aller plus loin. Et le serpent est puissant !

Voici de quoi donner un peu l'eau à la bouche, et ce n'est que le haut de l'iceberg !

4.1. La bibliothèque *numpy*

Nous l'avons déjà rencontrée avec la fonction `isprime()`.

La fonction `roots()`

Elle calcule les racines d'un polynôme, y compris les racines complexes...
Exemple d'un second et d'un troisième degré.

```
1  import numpy as np
2
3  a, b, c = -2.3, 7, 13
4  coefficients = [a, b, c]
5  racines = np.roots(coefficients)
6  print(racines)
7
8  a, b, c, d = 1, 0, -1, -1
9  coefficients = [a, b, c, d]
10 racines = np.roots(coefficients)
11 print(racines)
```

```
[ 4.34447962 -1.30100136]
[ 1.32471796+0.j         -0.66235898+0.56227951j -0.66235898-0.56227951j]
```

La fonction `complex()`

Permet de faire tous les calculs avec les nombres complexes.

```
1  import numpy as np
2
3  c = complex(0.1, -0.2)
4  z = complex(0, 0)
5  for k in range(1,1001) :
6    z = z**2 +c
7
8  print (np.abs(z))
```

On calcule ici le module d'un nombre complexe z après 1 000 itérations de la forme $z = z^2 + c$ pour $c = 0.1 - 0.2i$. Le résultat est 0.2297529205473612...
On en déduira que c appartient à l'ensemble de Mandelbrot, car le module est inférieur à 2.

Bonjour les fractales !

4.2. La bibliothèque *sympy*

Toute la panoplie du calcul symbolique.

La fonction `expand()`

Elle développe une expression algébrique ; il faut commencer par déclarer les symboles.

```
1 import sympy as sp
2
3 x = sp.symbols('x')
4 y = sp.symbols('y')
5 expression = (x - y)**4
6 developp = sp.expand(expression)
7 print(expression, "=", developp)
```

Voici la réponse : $(x-y)**4 = x**4 - 4*x**3*y + 6*x**2*y**2 - 4*x*y**3 + y**4$

La fonction `solve()`

Pour résoudre une équation du type $f(x) = 0$, mais aussi un système.

```
1 import sympy as sp
2
3 x = sp.symbols('x')
4 y = sp.symbols('y')
5 eq1 = sp.Eq(2*x + y, 5)
6 eq2 = sp.Eq(x - 3*y, -7)
7 sys_sol = sp.solve((eq1, eq2), (x, y))
8 print(sys_sol)
```

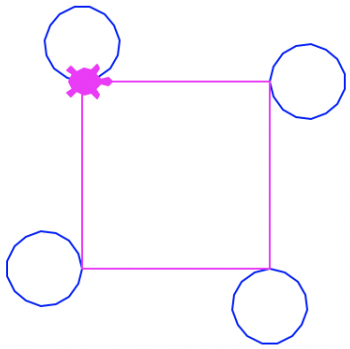
Sa réponse : $\{x : 8/7, y : 19/7\}$

Et il y a aussi limites, dérivées, intégrales, équations différentielles...

4.3. La bibliothèque *turtle*

C'est une bibliothèque standard de Python. Elle permet de programmer des dessins géométriques de la même manière que le langage LOGO et sa tortue. Ce sont d'ailleurs les mêmes instructions, mais en anglais.

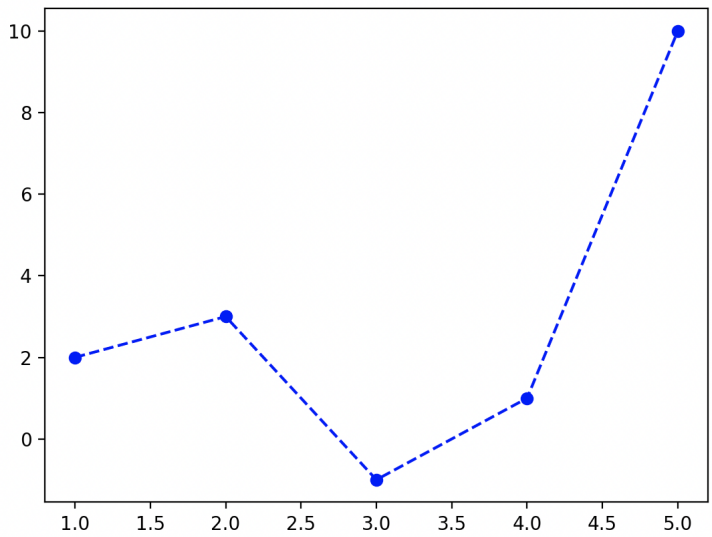
```
1  import turtle as t
2
3  ecran = t.Screen()
4  t.shape("turtle")
5  t.color("magenta")
6  for k in range(4):
7    t.forward(100)
8    t.right(90)
9    t.color("blue")
10   t.circle(20)
11   t.color("magenta")
12 ecran.mainloop()
```



4.4. La bibliothèque *mathplotlib* et le sous-module *pyplot*

Un ensemble de fonctions permettant de créer et personnaliser très rapidement des graphiques 2D (courbes, histogrammes, nuages de points, etc.).

```
1 import matplotlib.pyplot as plt
2
3 x = [1, 2, 3, 4, 5]
4 y = [2, 3, -1, 1, 10]
5
6 plt.plot(x, y, color="blue", linestyle="--", marker="o")
7 plt.show()
```



5. Pour apprendre... beaucoup plus

Gérard Swinnen, *Apprendre à programmer avec Python 3*, 2012.

On peut l'acheter aux éditions Eyrolles, environ 35 euros, mais on peut aussi le télécharger gratuitement et librement sur http://inforef.be/swi.python.htm