Nicolas Bourgeois

3e édition

Python

Du grand débutant à la programmation objet

Cours et exercices corrigés



ellipses

Références sciences

Python

Du grand débutant à la programmation objet

Cours et exercices corrigés

3e édition

Nicolas Bourgeois

ellipses

Collection Références sciences

dirigée par Paul de Laboulaye
paul.delaboulaye@editions-ellipses.fr

Retrouvez tous les livres de la collection et des extraits sur www.editions-ellipses.fr

ISBN 9782340-088580
© Ellipses Édition Marketing S.A., 2024
8/10 rue la Quintinie 75015 Paris

DANGER LE PHOTOCOPILLAGE TUE LE LIVRE

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5.2° et 3°a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective », et d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit constituerait une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

www.editions-ellipses.fr

Avant-propos

L'idée générale de ce livre est de procéder avant tout par l'exemple et la pratique. Aussi les concepts comme la syntaxe sont généralement introduits par un petit script commenté avant d'être définis proprement. De plus, chaque section contient de nombreux exercices dont les corrigés sont donnés en fin d'ouvrage. Le code est séparé du corps du texte par un cadre sombre :

```
print('Hello World')
```

Le résultat de son exécution est quant à lui précédé de trois chevrons (>>>) pour reproduire l'apparence qui vous sera bientôt familière de la console.

```
>>>
Hello World
```



Ce livre a été conçu pour être accessible à toute personne n'ayant jamais programmé, c'est pourquoi certaines étapes initiales sont très détaillées. Cependant, nous avons aussi pensé aux étudiant-e-s ayant déjà une expérience d'un autre langage comme C, PHP, Java, R ou encore Scilab. Pour ceux et celles-ci, nous avons ici et là ajouté quelques encarts (dans un cadre gris clair, avec un symbole d'engrenage, comme ce paragraphe) essayant d'expliquer les différences entre le langage auquel ils et elles sont habitué-e-s, et le fonctionnement de Python. Les lecteurs et lectrices débutant la programmation peuvent sereinement ignorer tous ces blocs précédés d'un engrenage.

Les exercices ne nécessitent pas pour être résolus de connaissance mathématique au-delà d'un baccalauréat scientifique ; en revanche il est probable que ceux des tout premiers chapitres intéresseront davantage des étudiant-e-s ayant un peu de familiarité avec l'arithmétique et la combinatoire. Au fur et à mesure de votre progression, vous verrez les mathématiques disparaître car les objets informatiques manipulés commencent à être suffisamment intéressants en eux-mêmes.

Il y a deux catégories d'exercices : les exercices d'application directe, dispersés au fil du cours, et les problèmes plus difficiles, récapitulés à la fin de chaque section. Ceci étant, la difficulté des exercices d'application va un peu croissante avec le livre, tenant compte du fait que vous progressez – essentiellement, vous êtes capables de manipuler un code de taille plus importante.

Les chapitres 1 à 3 contiennent la base que doit connaître toute personne souhaitant acquérir rapidement des bases en Python, quelle que soit sa motivation. Les chapitres 4 à 6 sont plutôt destinés à ceux et celles aspirant à un métier dans l'informatique, car les principes qui y sont développés restent pertinents quel que soit le langage. Les chapitres 7 et 8 peuvent intéresser tout le monde, mais seront vitaux pour les futurs data scientists. Enfin, les deux derniers chapitres introduisent l'objet, notion fondamentale pour l'informatique de gestion.

Cette troisième édition a été l'occasion de vérifier la compatibilité du code avec les versions les plus récentes de Python (3.12 et 3.13) et de se rapprocher des conventions d'écriture définies par la PEP 8 afin d'assurer la plus grande lisibilité possible des exemples et corrigés d'exercices.

Table des matières

1 Découvrir le langage **7**
- 1.1 Installer et exécuter Python 7
- 1.2 Utiliser Python comme calculatrice 9
- 1.3 Structures de contrôle 13
- 1.4 Résumé 18
- 1.5 Exercices plus avancés 18

2 Une première structure : la liste **21**
- 2.1 Définition 21
- 2.2 Opérations sur les listes 25
- 2.3 La mémoire et les listes 30
- 2.4 Résumé 32
- 2.5 Exercices plus avancés 33

3 Programmer avec des fonctions **35**
- 3.1 Définir et utiliser des fonctions 35
- 3.2 Utiliser les fonctions des librairies usuelles 40
- 3.3 Quelques aspects un peu plus subtils des fonctions 43
- 3.4 Résumé 48
- 3.5 Exercices plus avancés 50

4 Un peu d'algorithmique **53**
- 4.1 Algorithmes récursifs 53
- 4.2 Programmation dynamique 57
- 4.3 Exemple détaillé : trouver le plus court chemin 58
- 4.4 Résumé 63
- 4.5 Exercices plus avancés 63

5 Maîtriser le temps de calcul **65**
- 5.1 Quand l'optimisation devient nécessaire 65
- 5.2 La complexité au pire des cas 70
- 5.3 Méthodes de calcul 73
- 5.4 Résumé 77

5.5 Exercices plus avancés 78

6 Structures de données 79
6.1 Quelques alternatives à la liste 79
6.2 Le mapping 84
6.3 Bien choisir sa structure 86
6.4 Résumé 89
6.5 Exercices plus avancés 90

7 Analyser du texte 93
7.1 Conversion de chaînes 93
7.2 Rechercher et remplacer 94
7.3 Expressions rationnelles 97
7.4 Résumé 103
7.5 Exercices plus avancés 104

8 Importer et exporter des données 107
8.1 Manipulation de fichiers 107
8.2 Bases de données 113
8.3 Résumé 117
8.4 Exercices plus avancés 118

9 Créer des Classes en Python 121
9.1 Vers l'objet 121
9.2 Classes, constructeurs et méthodes 123
9.3 Séparation interface et contenu 125
9.4 Composer des objets, déléguer des méthodes 128
9.5 Résumé 134
9.6 Exercices plus avancés 134

10 Programmation orientée objet 137
10.1 L'encapsulation 137
10.2 L'héritage 141
10.3 Tout est objet 143
10.4 Résumé 147
10.5 Exercices plus avancés 148

11 Corrigés des exercices 149

Bibliographie 235

Index 237

Chapitre 1

Découvrir le langage

1.1 Installer et exécuter Python

Nous supposons que vous souhaitez commencer à programmer au plus vite sans vous préoccuper trop avant des questions d'environnement, c'est pourquoi nous commençons par quelques instructions qui pourront vous sembler un peu directives. Lorsque la question de choisir l'environnement de travail qui vous convient le mieux vous paraîtra pertinente, vous trouverez facilement sur le web un comparatif des différentes possibilités.

Avec un IDE

Si vous travaillez sur une machine pour laquelle vous disposez de droits d'administrateur, par exemple votre ordinateur personnel, nous vous recommandons d'installer successivement un interpréteur Python et un environnement de travail intégré (IDE).

Le premier est le minimum absolu pour pouvoir exécuter un script Python. Vous pouvez le télécharger sur le site de la *Python Software Foundation*. Les exemples de ce livre ont été vérifiés avec la version 3.12, mais nous vous recommandons d'installer la version stable la plus récente – il n'y aura pas de problème de rétrocompatibilité pour les versions 3.x entre elles.

Il est théoriquement possible de travailler avec le seul interpréteur et une console. Toutefois pour des raisons de confort nous recommandons d'installer également un IDE compatible avec votre système d'exploitation tel que *Thonny*, *Pyzo*, *PyCharm* ou encore *IDLE* – ce dernier étant fourni avec la distribution Python, mais pas nécessairement le plus facile à prendre en main. Vous disposerez ainsi d'une interface unique dans laquelle écrire votre code et lire les résultats, ainsi que d'un grand nombre d'autres fonctionnalités que vous découvrirez au fur et à mesure de votre progression.

Alternativement au fait d'installer sur votre machine l'interpréteur et l'IDE, vous avez la possibilité d'utiliser une interface en ligne, comme `pythonanywhere.com`, qui vous offre les fonctionnalités d'un IDE à distance.

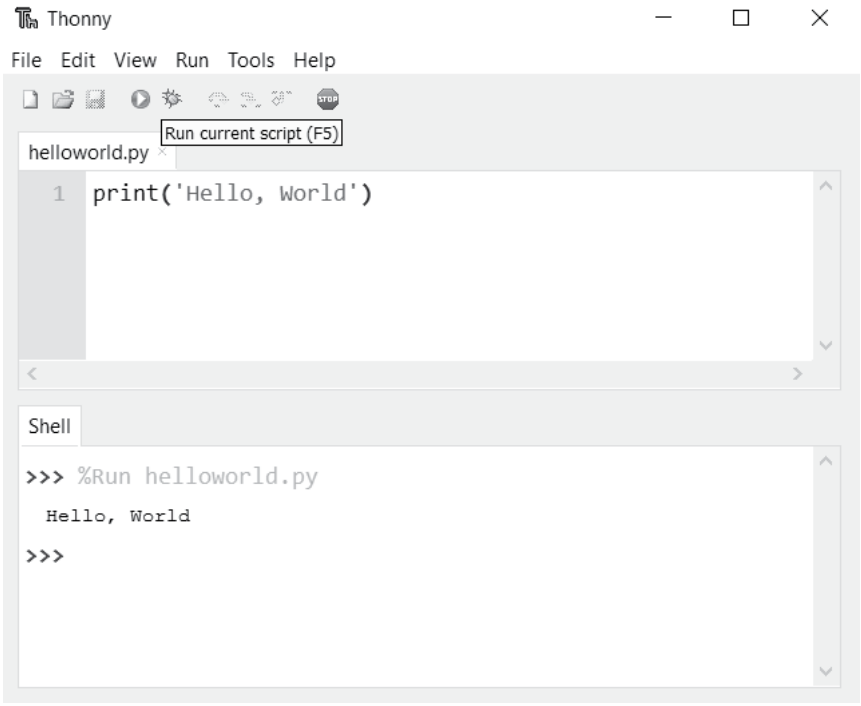


FIGURE 1.1 – Exemple d'interface (Thonny). Le code est écrit dans l'éditeur intégré et le résultat affiché dans la console (Shell) juste en-dessous.

Dans tout ce livre, nous ne ferons appel qu'à des librairies fournies dans la distribution de base, aussi nous n'aurons pas à nous préoccuper d'installer des modules complémentaires. Si vos centres d'intérêts vous amènent à avoir besoin de librairies externes, par exemple pour le calcul scientifique, vous pourrez vous référer à la documentation de *pip*.

Sans IDE

Si vous ne souhaitez pas ou ne pouvez pas installer d'IDE, par exemple parce que vous travaillez dans une salle informatique de votre université, c'est un tout petit peu plus compliqué, mais pas insurmontable. Il faut cependant au minimum qu'un interpréteur Python soit installé sur la machine [1].

Vous écrirez votre code dans un éditeur de texte quelconque, de préférence proposant une fonctionnalité de coloration syntaxique et ne générant pas de problème d'interprétation des caractères spéciaux. *Notepad++* sous Windows, *textwrangler*

1. Sachez en dernier recours que toute distribution Linux intègre un interpréteur Python, pas nécessairement la version la plus à jour cependant (typiquement, s'il s'agit d'une version de Python 2.x et non 3.x il vous faudra modifier un peu la syntaxe utilisée dans ce livre).

sous Mac OS ou encore *scribes*, *gedit* sous Linux sont de bons exemples. Vous enregistrerez ensuite votre fichier sous un nom de fichier portant l'extension .py. Ceci fait, vous ouvrirez un terminal et écrirez simplement `python xxx.py` où `xxx` est le nom de votre fichier, qui sera alors exécuté. Le retour sera affiché directement dans le terminal.

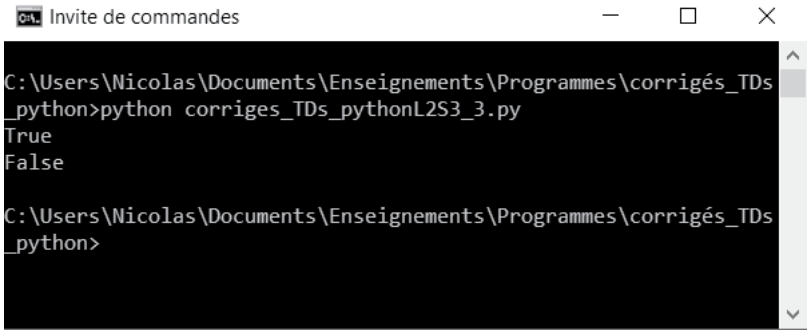


FIGURE 1.2 – Exemple de terminal (Windows).

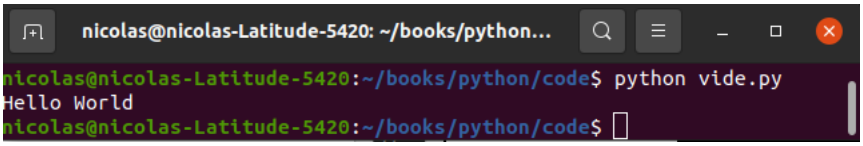


FIGURE 1.3 – Exemple de terminal (Ubuntu).

1.2 Utiliser Python comme calculatrice

Même dans l'hypothèse où vous n'avez aucune notion d'algorithmique, vous êtes certainement familier-ère avec le fonctionnement d'une calculatrice. Le moins qu'on puisse attendre de Python est qu'il vous permette d'en simuler une. Cela nous fournit l'occasion de notre premier script :

```
x = 3798
y = 48518
z = y * x
print(z)
```

Lequel produit dans la console le résultat suivant :

```
>>>
184271364
```

Les lignes une et deux affectent la valeur à deux variables, la troisième en effectue la multiplication, la quatrième utilise la fonction `print()` pour provoquer un affichage dans la console. Le même résultat aurait pu être obtenu en écrivant directement `print(3798 * 48518)`.



Si vous avez déjà programmé dans d'autres langages, vous remarquerez que la déclaration de variable est simplifiée à l'extrême : pas de mention de type ni d'allocation de mémoire, pas même de mot-clef indiquant la déclaration. Cela ne signifie pas pour autant que Python n'est pas un langage typé. Simplement, le type est résolu à partir du contexte.

```
x = 3798
y = 48518
z = y * x
print(type(z))
s = 'Hello World'
print(type(s))
```

```
>>>
<class 'int'>
<class 'str'>
```

Remarquez que Python attribue aux variables des types (ici `int` pour entier et `str` pour chaîne de caractères), lesquels peuvent être retrouvés à partir de la fonction `type()`. Ces types conditionnent le comportement des fonctions, opérateurs, etc. qui leur sont appliqués. Nous aurons amplement l'occasion de revenir sur cette notion au fur et à mesure de notre progression, mais voyons tout de suite un exemple très simple qui illustre combien il est important d'avoir cette idée de type en tête quand on programme.

```
x, y, z, t = 2, 1 / 3, '2', '1/3'
print(type(x), type(y), type(z), type(t))
print(x + y)
print(z + t)
```

```
>>>
<class 'int'> <class 'float'> <class 'str'> <class 'str'>
2.3333333333333335
21/3
```

À la ligne 3, nous effectuons une addition entre un entier et un décimal (`float`) et nous obtenons le résultat attendu – à part l'arrondi un peu bizarre. À la ligne 4, nous utilisons le même opérateur `+`, mais ici entre deux chaînes de caractères : le résultat obtenu est non pas une addition mais une concaténation. Si vous faites l'expérience, vous remarquerez enfin que tenter `x+z` provoque un affichage d'erreur :

```
>>>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Retenons donc de tout ceci une chose : un opérateur fonctionne différemment, voire pas du tout, selon le type des objets auxquels on l'applique. Il est donc important de distinguer le type des objets manipulés, par exemple l'entier `3` de la chaîne de caractères `'3'`.



Contrairement à certains autres langages comme PHP qui effectueraient une conversion implicite, Python explique qu'il ne sait pas comment interpréter cet opérateur entre deux objets pour lesquels le comportement prévu est radicalement différent : il préfère lancer une exception plutôt que de risquer la propagation d'un comportement non prévu. Toutefois lorsqu'un type est inclus dans un autre (`int` dans `float`), il effectue une conversion vers le type le plus général.

Pour le moment, nous n'avons utilisé que les opérations élémentaires, mais il est certain que Python peut également gérer des fonctions mathématiques plus complexes.

```
from math import *

print(sqrt(2), log(7.12), factorial(6))
```

```
>>>
1.4142135623730951 1.9629077254238845 720
```

Pas de surprise dans le comportement des fonctions racine carrée (`sqrt`), logarithme (`log`) et factorielle (`factorial`). Notez cependant la première ligne qui sert à importer ces fonctions depuis la librairie `math`, sans quoi le code renverrait un message d'erreur. Nous verrons un peu plus loin le fonctionnement des librairies, mais

l'idée ici est simple : seules quelques fonctions sont immédiatement disponibles pour Python, les autres doivent être explicitement importées depuis une librairie afin que l'interpréteur sache exactement ce qu'il doit faire.

Si la librairie `math` contient l'essentiel des fonctions mathématiques de base, elle ne saurait être exhaustive. Parfois vous aurez besoin de coder vous-même une fonction spécifique, parfois vous préfèrerez aller la chercher parmi le grand nombre de bibliothèques existantes sur internet. Si la seconde solution est souvent optimale en terme de temps de calcul et permet de limiter la taille de votre code, la première peut être plus pédagogique, plus modulable et vous faire économiser du temps de recherche. C'est un dilemme auquel vous serez souvent confronté-e.

Par exemple, si vous souhaitez calculer le nombre de combinaisons de p parmi n, il est plus économique de programmer vous-même :

```
from math import factorial

p, n = 7, 23
print(factorial(n) // (factorial(p) \
                       * factorial(n - p)))
```

```
>>>
245157
```

Exercice 1

Trouvez les solutions de $3x^2 - 7x = 23$.

Exercice 2

Sachant qu'il existe dans la librairie `math` une fonction `gcd(a,b)` qui renvoie le plus grand diviseur commun à `a` et `b`, trouvez la forme irréductible de la somme :

$$\frac{217}{440} + \frac{101}{256} + \frac{86}{71}$$

Pour clore cette première section, voici les opérateurs et fonctions mathématiques parmi les plus fréquemment utilisés.

Opérateurs sur les nombres

`+`	addition	`-`	soustraction	`*`	multiplication
`/`	division réelle	`//`	quotient de la division euclidienne	`%`	reste de la division euclidienne
`**`	puissance				

Fonctions mathématiques

`log`	logarithme	`exp`	exponentielle	`sqrt`	racine carrée
`floor`	partie entière	`gcd`	plus grand diviseur commun	`factorial`	factorielle
`cos`	cosinus			`abs`	valeur absolue

1.3 Structures de contrôle

Les conditions `if ... : else:`

Pour sortir de ce rôle de calculatrice et commencer à faire un peu d'algorithmique, nous allons maintenant introduire les conditions et les boucles. Afin de ne pas être trop théorique, commençons par un petit exemple :

```
s = input("Entrez un nombre pair : ")
n = int(s)
if n % 2 == 0:
    print(n // 2)
else:
    print("Le nombre n'est pas pair")
```

Si l'utilisateur entre un nombre pair, nous obtenons :

```
>>>
Entrez un nombre pair : 26
13
```

À l'inverse, s'il écrit un nombre impair :

```
>>>
Entrez un nombre pair : 33
Le nombre n'est pas pair
```

La première ligne du programme demande à l'utilisateur de rentrer un nombre ; le retour de l'utilisateur est toujours stocké *comme une chaîne de caractères* par Python, c'est pourquoi à la seconde ligne nous lui demandons explicitement de le convertir en entier. Si l'utilisateur avait rentré d'autres caractères, l'interpréteur aurait renvoyé un message d'erreur à ce moment.

L'instruction `if ...:` teste cette condition. Si elle est vraie, le bloc suivant est exécuté (ici la ligne 4), sinon il ne l'est pas et le bloc situé après le mot-clef `else:` est exécuté à la place (ici la ligne 6).

Ce qui détermine le bloc est l'indentation, c'est-à-dire l'espace situé au début de la ligne. Tous les éléments de même indentation font partie d'un même bloc, tandis qu'un retour à un niveau d'indentation inférieur indique que le bloc est terminé. Quant aux deux points (`:`) ils servent à indiquer la fin de la condition.

On peut mesurer le rôle de l'indentation avec l'exemple ci-dessous :

```
x = 3
if x > 5:
```

```
    x = x + 2
    x = x + 7
x = x + 1
print(x)
```

```
>>>
4
```

Comme la condition n'est pas vérifiée, les deux premières incrémentations, qui sont indentées et font donc partie du bloc conditionnel, sont ignorées. En revanche, la troisième, qui n'est pas indentée, ne fait pas partie de la condition et est donc exécutée indépendamment de celle-ci.



Là où l'indentation est dans beaucoup de langages une convention destinée à faciliter la lecture, elle est obligatoire en Python et se substitue aux accolades. Si cela peut sembler un peu contraignant, cela contribue à épurer la syntaxe, de même l'absence de points-virgules en fin d'instruction, puisque celles-ci sont simplement gérées par les retours à la ligne.

À noter que mot-clef `if` peut également être utilisé en ligne dans certains contextes, par exemple pour une affectation, comme dans l'exemple ci-dessous. C'est ce qu'on appelle parfois l'opérateur ternaire.

```
n = 67
suivant = n // 2 if n % 2 == 0 else 3 * n + 1
print(suivant)
```

```
>>>
202
```

La boucle `for ... in range()`

```
f = 1
for i in range(1, 11):
    f = f * i
print(f)
```

```
>>>
3628800
```

Vous l'aurez peut-être reconnu, ce programme calcule la factorielle de `10`, et donne donc rigoureusement le même résultat que `factorial(10)` – avec un temps de calcul plus élevé, ce qui le rend tout à fait inutile en pratique. Mais il constitue un exemple paradigmatique de boucle. Examinons-le de plus près.

La fonction `range(a,b)` énumère la suite des nombres de `a` (inclus) à `b` (exclu). `range(1,11)` va donc lister successivement `1,2,3,4,5,6,7,8,9,10`. La ligne 2 peut se traduire de la façon suivante : nous allons énumérer par ordre croissant les nombres compris entre 1 inclus et 11 exclu. La valeur de ces nombres successifs sera à chaque étape stockée dans la variable `i`, puis nous allons effectuer, toujours à chaque étape, la suite d'opérations figurant dans le bloc indenté suivant (en l'occurrence la ligne 3).

Pas à pas, la résolution de l'algorithme pourrait être décortiquée ainsi :

1. `f` prend la valeur `1`, on passe à la ligne 2
2. `i` prend la valeur `1`, on passe à la ligne 3
3. `f` prend la valeur `1*1=1`, on revient à la ligne 2
4. `i` prend la valeur `2`, on passe à la ligne 3
5. `f` prend la valeur `1*2=2`, on revient à la ligne 2
6. `i` prend la valeur `3`, on passe à la ligne 3
7. `f` prend la valeur `2*3=6`, on revient à la ligne 2, etc.

Exercice 3

Modifiez l'indentation du code de l'exemple précédent de façon à ce qu'il affiche toutes les valeurs de `factorial(x)` pour `x` de `1` à `10` et non pas uniquement `factorial(10)`.

Exercice 4

Affichez les nombres pairs inférieurs à 40.

Exercice 5

Affichez les 40 premiers nombres de la suite de Fibonacci définie par la récurrence $\Phi_0 = \Phi_1 = 1$ et $\Phi_{n+2} = \Phi_{n+1} + \Phi_n$.

Nous avons désormais vu une structure de condition et une structure de boucle. Il en existe bien d'autres que nous allons découvrir progressivement. Le plus souvent, vous les utiliserez de façon combinée, comme dans l'exemple suivant.

```
from math import sqrt, floor

n = 28928203297
for i in range(2, floor(sqrt(n)) + 1):
    if n % i == 0:
        print(n // i, i)
```

```
>>>
4132600471 7
8128183 3559
1161169 24913
```

Ce script va afficher l'ensemble des diviseurs de `28928203297` non-triviaux (c'est-à-dire autres que `1` et lui-même).

La boucle `for` de la ligne 3 examine successivement tous les nombres de 2 à $\sqrt{\texttt{n}}$. En effet, il n'est pas besoin d'examiner les nombres plus grands, puisque si `ab = n`, nécessairement l'un des deux est plus petit que $\sqrt{\texttt{n}}$. À chaque étape de la boucle, nous vérifions (ligne 4) que `n` est divisible par `i`. Si c'est le cas, la ligne 5 provoque l'affichage des deux diviseurs trouvés. Notez la double indentation du `print()` qui indique qu'il se trouve dans le `if`, lui-même inclus dans le `for`.

Exercice 6

Affichez les nombres compris entre 1 à 100 et divisibles par 3, 5 ou 7.

La boucle `while`

Troisième et dernière structure de contrôle que nous allons voir dans cette introduction au langage, la boucle `while` sert à répéter une action non pas un nombre fixé de fois, mais tant qu'une certaine condition reste valide.

```
m = 13
while m > 1:
    if m % 2 == 0:
        m = m // 2
    else:
        m = 3 * m + 1
    print(m)
```

```
>>>
40
20
10
5
16
8
4
2
1
```

À chaque itération de la boucle, on commence par regarder si le nombre `m` est plus grand que `1`. Si ce n'est pas le cas, la boucle s'arrête, sinon on examine deux possibilités : si `m` est pair, on divise par deux, s'il est impair on multiplie par trois et ajoute un. Notez qu'en théorie, il est tout à fait possible que cet algorithme ne termine jamais puisqu'il et difficile de prévoir si `m` va augmenter à l'infini ou redescendre finalement vers `1`. Cependant, vous aurez peut-être reconnu la suite de Syracuse, un célèbre problème ouvert en mathématique : on ne connaît à ce jour pas de nombre `m` tel que l'algorithme ne termine pas, et on conjecture qu'il n'en existe pas.



On pourrait être tenté d'objecter que la boucle `for` n'est guère plus qu'un cas particulier de la boucle `while`. Toutefois l'utilisation de `for` avec `range()` n'est qu'une application spécifique d'un outil d'énumération puissant dont les possibilités se révèleront à mesure que nous étudierons les structures de données itérables.

Exercice 7

Réécrivez le code qui affiche les diviseurs de `28928203297` en utilisant une boucle `while` au lieu de `for`.

Exercice 8

Déterminez si `4132600471` est un nombre premier, c'est-à-dire qu'il ne possède d'autre diviseur que `1` et lui-même.

Exercice 9

Affichez les nombres premiers compris entre 1 et 100.

1.4 Résumé

Python peut stocker, manipuler et afficher des variables. Ces variables ont un type, qui affecte le comportement des opérateurs et des fonctions qui les manipulent.

Python peut être utilisé pour effectuer des opérations mathématiques, dont la plupart doivent être importées depuis la librairie math, en utilisant la syntaxe `from math import`.

Le cœur de nombreux programmes est constitué par des structures de contrôle qui permettent d'automatiser certaines opérations algorithmiques :

— `if ... : else:` permet de tester une condition et de décider d'effectuer telle ou telle action en fonction du résultat,

— `for i in range(x):` permet d'effectuer `x` actions successives, qui peuvent dépendre du nombre `i`, lequel varie de `0` à `x-1`,

— `while ... :` permet d'effectuer certaines actions de façon répétée tant qu'une certaine condition continue d'être vérifiée,

1.5 Exercices plus avancés

Problème 1 *(Écriture binaire)*

a) Étant donné un chiffre `i` et un nombre entier `n`, écrivez un programme qui affiche le `i`-ème chiffre de `n`. Par exemple, si `n=375` et `i=0` on doit afficher `5`. On pourra utiliser des divisions euclidiennes.

b) Modifiez le programme précédent de façon à récupérer le `i`-ème chiffre de `n` selon l'écriture binaire (et non décimale). Par exemple, comme `17` s'écrit `10001` en binaire, son deuxième chiffre est `0`.

c) Écrivez un programme qui renvoie l'écriture binaire complète du nombre décimal `41819` et l'écriture décimale complète du nombre binaire `1001010000110101`.

Problème 2 *(Indicatrice d'Euler)*

Contexte : le chiffrement RSA est l'un des systèmes de protection des données les plus populaires. Il est basé sur un outil arithmétique, l'indicatrice d'Euler. En résumé, l'idée est qu'il est possible grâce à cet outil de trouver deux nombres a_n, b_n tel que $\forall x, x^{a_n b_n} = x$ modulo n. On chiffre alors le message (exprimé comme une suite de nombres) en passant chaque nombre à l'exposant a_n et on déchiffre en passant le résultat à l'exposant b_n. Dans ce problème on s'intéresse juste au calcul de l'indicatrice, et on effectuera le chiffrement lui-même dans le problème suivant.

L'indicatrice d'Euler $\phi(n)$ associe à un nombre n le nombre d'entiers i strictement inférieurs à n qui sont premiers avec n, c'est-à-dire :

$$\phi(n) = |0 < i < n, \gcd(i, n) = 1|$$

a) Proposez un programme qui à l'aide d'une boucle `for` calcule directement la valeur de l'indicatrice de `n` pour un `n` donné.

On peut prouver que si la décomposition de `n` en facteurs premiers s'écrit $n = \prod p_i^{a_i}$ alors $\phi(n) = \prod p_i^{a_i - 1}(p_i - 1)$.

b) Proposez un programme, qui à l'aide de deux boucles `while` imbriquées, énumère les facteurs premiers d'un nombre, avec la bonne multiplicité. Par exemple, pour `n=54`, il devra afficher :

```
>>>
2
3
3
3
```

c) Modifiez le programme précédent de façon à calculer la valeur de ϕ`(n)` pour un `n` donné, en utilisant la décomposition en produit ci-dessus.

d) Vérifiez sur de grands nombres que cette deuxième méthode est beaucoup plus rapide.

e) Soient `p` et `q` deux nombres premiers. Vérifiez que leur indicatrice d'Euler est égale à `(p-1)(q-1)`.

Problème 3 *(Chiffrement RSA)*

Ce problème est la suite du problème 2, mais peut être résolu indépendamment.

a) On pose `p = 8191` et `q = 127`. Calculez `phi = (p-1)(q-1)`

b) Trouvez deux nombres entiers `a,b` tels que :
— `a` est premier avec `phi`,
— `a*b % phi = 1`.
Vous pouvez soit faire par énumération des possibles jusqu'à en trouver un, soit utiliser l'algorithme d'Euclide étendu si vous le connaissez.

c) Traduisez une lettre sous forme de nombre en utilisant la méthode `ord()` qui renvoie le code ASCII du caractère passé en argument. Chiffrez le nombre obtenu en le passant à l'exposant `a` modulo `n`.

d) Vérifiez qu'on peut bien déchiffrer le nombre obtenu en le passant à l'exposant `b` modulo `n`, puis retrouvez la lettre originale grâce à la fonction `chr()` qui renvoie le caractère ASCII dont le numéro correspond au nombre passé en argument.

Remarque : même si on le fait ici pour le besoin de l'exercice, dans la pratique, on ne chiffre surtout pas lettre par lettre avec RSA, car ce serait un chiffrement très vulnérable. Il suffirait par exemple de lister les 26^2 *possibilités ou de repérer les mots les plus fréquents pour briser le chiffrement. On codera plutôt des blocs de taille arbitraire.*

Chapitre 2

Une première structure : la liste

2.1 Définition

Pour le moment nos programmes calculent beaucoup d'expressions, mais se contentent de les afficher. Rapidement, le besoin se fait sentir de stocker de grandes quantités de variables ou de résultats. C'est là qu'interviennent les listes.

```
t = [1, 2, 7.03, "Bonjour", -9]
print(t[2])
t[1] = 6
print(t)
```

```
>>>
7.03
[1, 6, 7.03, 'Bonjour', -9]
```

À la ligne 1, nous déclarons une liste de la façon la plus simple du monde : en énonçant l'ensemble de ses éléments, séparés par des virgules, le tout entre crochets. Nous pouvons ensuite accéder aux éléments aussi bien en lecture (ligne 2) qu'en écriture (ligne 3) en utilisant leur indice, c'est-à-dire leur position dans la liste. Attention, Python compte les éléments à partir de `0` : l'élément `t[0]` est `1`, l'élément `t[1]` est `2`, l'élément `t[2]` est `7.03`, l'élément `t[3]` est `'Bonjour'` et l'élément `t[4]` est `-9`.

Il est également possible d'accéder aux éléments en comptant depuis la fin : `t[-1]` renvoie le dernier élément, `t[-2]` l'avant-dernier, etc.



À l'opposé d'un tableau Java, déclaration d'une liste en Python ne suppose aucune mention de taille ou de type : la longueur n'est pas fixe, et elle peut contenir plusieurs types de variables. Nous verrons au chapitre 6 comment créer des collections immuables et/ou de type fixe (la classe `str` en est l'exemple le plus immédiat), et quand cela est ou non plus pertinent que l'emploi d'une liste.

Exercice 10

Créez la liste des nombres entiers de 1 à 10, puis remplacez tous les nombres pairs par leur moitié.

Toutefois, le plus souvent, les listes sont construites dynamiquement, par exemple en utilisant une boucle. L'exemple suivant construit la liste des combinaisons de p parmi n :

```
from math import factorial

n = 9
nCp = []
for p in range(n + 1):
    x = factorial(n) // (factorial(p)
    * factorial(n - p))
    nCp.append(x)
print(nCp)
```

```
>>>
[1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
```

Plusieurs remarques de syntaxe sur cet exemple. Premièrement, lorsque nous écrivons `range(a)`, Python l'interprète comme étant `range(0,a)`. Surtout, le point important est constitué par l'expression `nCp.append(x)`. Son effet est d'ajouter une case *à la fin* de la liste `nCp` et d'insérer la valeur `x` dans cette case. Ainsi notre liste, initialement vide, grossit-elle d'un élément à chaque itération de la boucle.

Exercice 11

Modifiez le code précédent de sorte qu'il n'ajoute `x` à la liste que si `x` est pair.

Il serait également possible d'ajouter les éléments à d'autres endroits. La méthode `insert(i,x)` insère la valeur `x` en position `i` et décale tous les éléments situés après `i` d'une case pour lui faire la place. Voyez plutôt l'exemple ci-dessous :

```
liste = []
for i in range(5):
    liste.append(i)
for i in range(5, 10):
    liste.insert(0, i)
print(liste)
```

```
>>>
[9, 8, 7, 6, 5, 0, 1, 2, 3, 4]
```

Exercice 12

Créez la liste des nombres entiers de 1 à 100, et insérez-y les nombres de la forme $k \times \sqrt{2}, 1 \leq k \leq 60$ de façon à ce que la liste finale soit toujours ordonnée.

Python permet enfin une autre forme de déclaration de liste, plus concise que de manipuler une lourde boucle, plus proche de l'écriture mathématique d'une fonction, et également plus rapide à l'exécution :

```
puissances = [2 ** p for p in range(10)]
print(puissances)

partiel = [2 ** p for p in range(10) if p % 3 != 0]
print(partiel)
```

```
>>>
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
[2, 4, 16, 32, 128, 256]
```

La syntaxe `[f(x) for x in s]` permet de construire la liste de toutes les valeurs de la fonction `f` sur l'ensemble `s`. Cet ensemble peut être défini de plusieurs façons : dans le premier exemple c'est une simple succession de nombres entiers, mais dans le second nous rajoutons une condition supplémentaire avec un `if` – que l'exposant ne doit pas être divisible par 3.

Précisons qu'il est possible de multiplier les conditions et les itérations en ligne, comme dans l'exemple ci-dessous.

```
l = [x + y / 10 for x in range(4)
     if x % 2 == 0 for y in range(3)]
print(l)
```