Petite bible du langage Python

Antoine Royer et Loris Delafosse

2021 – 2022

Table des matières

Chapitre 1

Variables, test et boucles

1.1 Variables

1.1.1 Notion de variable

Une variable est un morceau de mémoire auquel le programmeur attribue un nom et une valeur susceptible d'être modifiée lors de l'exécution du programme.

Chaque variable est associée à un type, qui correspond au type de valeur que la variable peut contenir :

nom du type Python	description en français
`int`	un nombre entier
`float`	un nombre réel
`str`	une chaîne de caractères
`list`	une liste de variables
`bool`	un booléen

Le type va notamment conditionner les différentes opérations autorisées sur la variable. On ne peut pas, par exemple, comparer un entier et une chaîne de caractères. De même certains opérateurs (+, -, *...) n'ont pas le même sens suivant le type de variable sur lequel il s'applique.

1.1.2 Initialiser une variable

Initialiser une variable, c'est lui attribuer son nom et sa valeur initiale : `nom = valeur` avec :

nom le nom de la variable (peut contenir des lettres majuscules, minuscules et des chiffres. Attention à ne pas placer de chiffre en première place[1].) Il est recommendé de nommer les variables d'une manière qui renvoie explicitement à leur contenu dans le programme. (Les noms comme `compteur` ou `liste1` sont ainsi à préférer à `k` ou `a`.)

valeur Python détecte automatiquement le type de la variable comme étant le type de la valeur associée.

Lorsque l'on a plusieurs variables à initialiser simultanément, on peut passer par différentes syntaxes :

— si toutes les variables ont la même valeur initiale : `var_1 = var_2 = ... = valeur`

— si les variables ont des valeurs différentes :
`var_1, var_2, ... = valeur_1, valeur_2, ...`

1. `NomDeVariable123` est valide. `123variable` n'est pas valide car le nom commence par un chiffre. À noter qu'on peut également utiliser le symbole `_` dans les noms de variables.

Exemples d'initialisation sur des types usuels

Nous nous intéressons ici à l'initialisation de la variable `ma_variable` (notez l'originalité dans le choix du nom !)

Les entiers (ou réels) `ma_variable = 2` ou `ma_variable = 2.5` pour un `float`

Les chaînes de caractères `ma_variable = "Coucou"` L'important est de placer le texte entre guillemets.

Les listes `ma_variable = [1, 2, 3]` Les éléments sont à placer entre crochets. Les éléments d'une liste peuvent être de type différents : `ma_variable = [1, 2, False, "Coucou", [3, 4]]` est valide.

1.1.3 Calculs élémentaires sur les variables

Python gère beaucoup de types de variables, et il est possible d'en créer. Les chaînes de caractères font l'objet d'une section à part (voir 3.1), de même que les listes (voir 3.2).

	int	str	list
+	addition	concaténation	concaténation
-	soustraction	*non défini*	*non défini*
*	multiplication	répétition	répétition
**	puissance	*non défini*	*non défini*
/	division	*non défini*	*non défini*
//	quotient de la division euclidienne	*non défini*	*non défini*
%	reste de la division euclidienne	*non défini*	*non défini*

Lorsque l'on effectue un calcul, il est souvent intéressant de sauver le résultat dans une variable. On utilise alors la même syntaxe que pour l'initialisation :

```
>>> a = 2
>>> b = 5
>>> c = a + b
>>> c
7
```

Il existe également des syntaxes abrégées :

syntaxe complète	syntaxe abrégée
`var = var + a`	`var += a`
`var = var - a`	`var -= a`
`var = var * a`	`var *= a`
`var = var / a`	`var /= a`
`var = var // a`	`var //= a`
`var = var % a`	`var %= a`

Quelques exemples :

```
>>> a = 5
```

```
>>> a += 1
>>> a
6
>>> a *= 2
>>> a
12
>>> a /= 4
>>> a
3.0
>>> a **= 2
>>> a
9.0
```

1.1.4 Transtypage

Principe

En Python, il est possible d'effectuer du transtypage, i.e. de modifier le type d'une variable. Il suffit pour cela d'appeler la fonction qui porte le nom du type que l'on souhaite (voir les exemples ci-dessous) et de lui donner la variable à transtyper en argument.

Exemples

Obtenir un entier `variable = int(variable)` (initialement, `variable` doit être de type `float` ou `str`.)

Obtenir une liste `variable = list(variable)` (initialement, `variable` doit être de type `int` ou `str`.)

Obtenir une chaîne de caractères `variable = str(variable)` (initialement, `variable` doit être de type `int` ou `list`.)

1.2 Interactions : gestion des entrées sorties

1.2.1 Afficher du texte

On utilise la fonction `print` pour afficher du texte. On peut afficher :

— du texte : `print("du texte à afficher")`

— une variable : `print(ma_variable)`

— les deux en même temps : `print("la variable vaut :", ma_variable, "du texte .")` (il faut séparer le texte des variables par des virgules).

1.2.2 Récupérer une valeur

Il est parfois intéressant de récupérer une valeur auprès de l'utilisateur. On utilise alors la fonction `input`. Cette fonction peut prendre en argument une chaîne de caractères et renvoie une chaîne de caractères.

Tout cela sera développé ultérieurement.

1.3 Test conditionnel

1.3.1 Principe

Il s'agit d'une simple implication logique. Si une condition est vérifiée, alors on effectue telles actions. La syntaxe est :

```
if <condition>:
    <actions>
<autre action sans rapport avec le if>
```

Juste après la condition, il faut impérativement mettre un :. De manière plus générale, en Python, les doubles-points : indiquent le début d'un nouveau bloc (test conditionnel, boucle itérative, conditionnelle, fonction...)

Il faut bien avoir en tête que le niveau d'indentation détermine la portée du test (c'est valable pour les tests conditionnels, les boucles, les fonctions...) Autrement dit, tout ce qui suit le `if` avec un niveau d'indentation supplémentaire, correspond à ce qu'il faut faire si la condition est réalisée.

Si la condition du `if` n'est pas vérifiée, on peut effectuer d'autres actions grâce au mot-clef `else` :

```
if <condition>:
    <actions>
else:
    <actions>
```

On peut aller encore plus loin : si la condition 1 est vérifiée, on effectue telles actions, sinon, si la condition 2 est vérifiée alors on effectue telles actions... Et si aucune condition n'est vérifiée alors on effectue encore d'autres actions. La syntaxe pourrait être :

```
if <condition 1>:
    <actions>
else:
    if <condition 2>:
        <actions>
    else:
        if ...
```

Ce qui est très lourd. On utilisera plutôt le mot-clef `elif` qui remplace le `else` et le `if` qui se suivent. La syntaxe devient alors :

```
if <condition 1>:
    <actions 1>
elif <condition 2>:
    <actions 2>
else:
    <actions 3>
```

Pour expliquer un peu plus : si la <condition 1> est vraie, on effectue <actions 1>, si la <condition 1> est fausse et que la <condition 2> est vraie, on effectue <actions 2>. Et si les deux conditions sont fausses, on effectue <actions 3>.

1.3.2 Retours sur les booléens et les conditions

Le type booléen en Python est un type qui ne peut prendre que deux valeurs : `True` ou `False`. Une expression logique est vraie si elle est égale au booléen `True`.

Une expression logique est une phrase constituée de variables, d'opérateurs de comparaisons (ils sont à connaître), et potentiellement d'opérateurs logique (on les croise plus rarement, mais ça peut être utile de voir leurs rôles). Si la phrase est vraie (donc égale à `True`), la condition est vérifiée, donc le code précisé dans le bloc suivant le `if` est exécuté.

Opérateurs de comparaisons	
==	est égal à
!=	est différent de
>	est strictement supérieur à
>=	est supérieur ou égal à
<	est strictement inférieur à
<=	est inférieur ou égal à

Opérateurs logiques	
and	et logique
or	ou logique (inclusif)
not	complémentation

Attention à ne pas confondre l'opérateur d'affectation =, qui permet d'initialiser une variable, avec la comparaison "est égal à", ==.

1.3.3 Exemples

Expressions logiques

Il s'agit ici plus de logique que de programmation Python :

```
>>> a = 2
>>> b = 5
>>> a == b
False
>>> a < b
True
>>> (a < b) or (a == b)
True
>>> not ((a % 2) == (5 % 2))
True
```

Juste pour revenir sur la dernière expression logique, % est le reste de la division euclidienne par deux. Ainsi, la phrase : `(a % 2) == (b % 2)` est fausse, la complémentation la rend vraie.

Exemples

```
>>> a = 2
>>> b = 5
>>> if a == b:
...     a = a + 1
...
>>> a
2
>>> if a <= b:
...     a += 1 # une autre syntaxe pour a = a + 1
```

```
...
>>> a
3
```

On peut tout à fait faire un petit script :

```
# On demande l'âge et on stocke la chaîne de caractères
# renvoyée dans 'age'
age = input("Quel est votre âge ? ")

# On convertit la chaîne de caractères en un entier
age = int(age)

# Si la variable age est supérieure ou égale à 18
if age >= 18:
    print("Vieux machin !")

# Sinon (si la variable 'age' est strictement inférieure à 18)
else:
    print("Alcool interdit...")
```

Un autre exemple avec un `elif` :

```
nombre = int(input("Entrez un nombre : "))

if nombre == 5:
    print("Vous avez gagné !")

# Si la variable 'nombre' est différente de 5 et égale à 10
elif nombre == 10:
    print("Presque...")

# Si la variable 'nombre' est différente de 5 et différente de 10
else:
    print("Mmm... perdu...")
```

1.4 Boucle itérative

1.4.1 Principe

On cherche à présent à répéter les mêmes opérations un nombre déterminé de fois. On utilise généralement la syntaxe :

```
for i in range(n):
    <actions>
<action sans rapport avec la boucle>
```

On appelle `i` variable itératrice. On peut lui donner n'importe quel nom, et même, dans le cas où on ne compte pas se servir de la variable dans la boucle, on peut noter : `for _ in range(...):`. Il s'agit d'une variable liée, ce qui signifie qu'elle n'a pas besoin d'être initialisée auparavant et qu'elle n'a pas de sens en-dehors de la boucle.

En Python, il y a plusieurs manières d'utiliser `range` :

— en précisant uniquement la borne haute : `range(n)` (la variable itératrice va aller de 0 à n - 1)

— en précisant la borne basse et la borne haute : `range(bas, haut)` (va aller de `bas` à `haut - 1`)

— en précisant les bornes basses et hautes ainsi que le pas (la valeur dont la variable itératrice est incrémentée à chaque tour) : `range(bas, haut, pas)`

Comme pour le test conditionnel, les actions à effectuer sont placées dans un bloc indenté. Ce dernier commence par `:`, et se termine avec la fin du niveau d'indentation.

On peut également utiliser les boucles itératives autrement. L'idée est que la variable itérative prenne pour valeur les éléments d'une liste ou les lettres d'une chaîne de caractères. En notant `ma_variable` une liste ou une chaîne de caractères, la syntaxe devient :

```
for i in ma_variable:
    <actions>
```

1.4.2 Exemples simples

Prise en main de `range`

```
>>> for i in range(3):
...     print(i)
...
0
1
2
>>> for i in range(1, 3):
...     print(i)
...
1
2
>>> for i in range(0, 10, 2):
...     print(i)
0
2
4
6
8
```

Boucler sur une liste

```
>>> ma_liste = [1, 2, "Coucou", [3, 4]]
>>> for element in ma_liste:
```

```
...     print(element)
...
1
2
Coucou
[3, 4]
```

Boucler sur une chaîne de caractères

```
>>> ma_chaine = "Bonjour"
>>> for lettre in ma_chaine:
...     print(lettre)
...
B
o
n
j
o
u
r
```

1.4.3 Exemples plus avancés

Pour avoir les points d'une fonction $y = x^2$, on peut utiliser une boucle itérative qui, pour chaque valeur de x associe la valeur de y :

```
>>> x = [0, 1, 2, 3, 4, 5] # les abscisses des points
>>> y = [] # les ordonnées
>>> for abscisse in x: # pour chaque x
...     y.append(abscisse ** 2) # on ajoute les éléments de x au
 ↪  carré à la liste y
...
>>> y
[0, 1, 4, 9, 16, 25]
```

On peut également avoir à chercher un élément dans une liste. Ci-dessous, on cherche l'élément 1 dans la liste `[1, 2, 3, 4, 5]` :

```
>>> ma_liste = [1, 2, 3, 4, 5]
>>> for element in ma_liste:
...     if element == 1:
...         print("Trouvé")
...
Trouvé
```

1.5 Boucle conditionnelle

1.5.1 Principe

La boucle conditionnelle intervient à la frontière entre le test conditionnel et la boucle itérative. On cherche à répéter un certain nombre d'actions tant qu'une condition donnée est vraie. Nous ne reviendrons pas ici sur les conditions et les booléens (voir 1.3.2).

La syntaxe est la suivante :

```
while <condition>:
    <actions>
<actions sans rapport avec la boucle>
```

On note ici aussi l'importance du : qui signifie l'entrée dans le bloc et la fin du niveau d'indentation qui en signal la fin.

1.5.2 Exemples

Il s'agit ici d'un petit script qui affiche le plus grand nombre entier tel que son carré soit inférieur ou égal à 25.

```
i = 20
# tant que i ** 2 est plus grand que 25 ...
while i ** 2 > 25:
    i -= 1 # ... i prend la valeur i - 1
print(i)
```

Attention : si la condition est toujours vérifiée, alors le programme ne s'arrêtera jamais ! Au contraire, si la condition n'est pas vérifiée, les actions dans la boucle ne seront jamais traitées.

Par exemple, ce petit script affiche 4 en sortie.

```
i = 4
while i ** 2 > 25:
    i -= 1
print(i)
```

Chapitre 2

Fonctions et modules

2.1 Fonctions

2.1.1 Principe d'une fonction

Une fonction est un ensemble d'instructions qui prend des paramètres (ou arguments) en entrée et qui renvoie quelque chose (possiblement rien, mais ne rien renvoyer, c'est renvoyer quelque chose : rien). Une fonction est définie par `def nom_de_ma_fonction():`. Tout le code qui est dans la fonction doit être indenté. A la fin de la fonction, on peut renvoyer une variable grâce à `return` :

```
def ma_fonction():
    <actions dans la fonction>
    return variable_a_renvoyer
<actions en dehors>
```

Les arguments

Les arguments sont des variables que l'on donne à la fonction en entrée.

Les arguments sont précisés entre les parenthèses, chaque argument est séparé par une virgule : `def ma_fonction(arg_1, arg_2, ...):`. Il faut avoir à l'esprit que les arguments de la fonction sont internes à la fonction, on parle de variables locales (ou liées) (par opposition aux variables globales qui sont définies en dehors de toute fonction). Par exemple :

```
1 # retourne le carré de 'nombre'
2 def carre(nombre):
3     return nombre ** 2
4
5 print(carre(2))
6 print(nombre)
```

Ainsi la ligne 5 affiche 4 et la ligne 6 renvoie une erreur qui stipule que la variable `nombre` n'est pas définie.

De même, lorsque l'on passe une variable en argument à une fonction, la fonction ne modifie pas la variable, elle ne travaille que sur une copie qui disparait à la fin de la fonction :

```python
a = 2
def fonction(b):
    return b + 2

print(fonction(a)) # affiche 4
print(a) # a vaut toujours 2
```

Pour modifier la variable donnée en argument, il faut affecter à la variable l'appel de la fonction :

```python
a = 2
def fonction(b):
    return b + 2

a = fonction(a)
print(a) # affiche 4
```

Cas particulier : les listes

La seule exception à ce qui précède est les listes (voir 3.2). Si l'on donne en argument à une fonction une liste et que cette fonction modifie la liste, alors la liste est modifée même en-dehors de la fonction. On dit alors que la liste est modifée par effet de bord.

```python
liste = [1, 2, 3]
def fonction(une_liste):
    une_liste[0] = une_liste[-1]

print(fonction(liste))
print(liste)
```

La ligne 5 affiche `None` (car la fonction ne renvoie rien). Et la ligne 6 affiche `[3, 2, 3]`, la liste a bien été modifiée.

Retourner un résultat

Une fonction renvoie toujours quelque chose. Si on ne précise pas quoi renvoyer (avec `return`), alors la fonction renvoie "rien", ce qui en Python se dit `None`.

2.1.2 Manipulations avancées sur les arguments de fonction

En Python les arguments d'une fonction sont séparés en trois types :

- Les arguments positionnels qui ne sont interprété par la fonction que par leur position lors de l'appel à la fonction.
- les arguments par mot-clef qui sont interprété uniquement par leur nom.
- les arguments les plus courants qui supportent les deux types précédent.

À la fin de cette partie, une courte section sera dédiée aux arguments optionnels qui peuvent être des trois types ci-dessus.

Arguments positionnels

Les arguments positonnels (et seulement positionnels) doivent être donnés dans le même ordre que celui utilisé lors de la définition de la fonction. Pour les utiliser, il faut mettre un / dans les arguments de la fonction après les arguments positionnels :

```
def fonction(arg1, arg2, /):
    print(arg1, arg2)

fonction(1, 2) # affiche "1 2"
fonction(1, arg2=2) # renvoie une TypeError
```

Lors de l'appel à la fonction, il n'est donc pas possible de passer un argument par son nom et la seule manière de passer cet argument à la fonction est de le passer dans l'ordre défini. Il est possible de donner un nombre arbitraire d'argument positionnels grâce à la syntaxe :

```
def fonction(*args):
    print(args)

fonction(1, 2, 3) # affiche "(1, 2, 3)"
fonction(1, "texte", [1, 2, 3]) # affiche "(1, 'texte', [1, 2,
 ↪  3])"
```

Argument par mot-clef

À l'inverse des arguments positionnels, les arguments par mot-clef ne peuvent être passés que s'ils sont précédés par leur nom lors de l'appel à la fonction. L'ordre dans lequel ces arguments sont passés n'a aucune importance. Pour forcer des arguments en arguments par mot-clef seuleument, il faut utiliser la syntaxe :

```
def fonction(*, arg1, arg2):
    print(arg1, arg2)

fonction(1, 2) # renvoie une TypeError
fonction(arg1=1, arg2=2) # affiche "1 2"
fonction(arg2=2, arg1=1) # affiche "1 2"
```

À l'instar des arguments positionnels il est possible de passer un nombre arbitraire d'arguments par mot-clef à une fonction grâce à la syntaxe :

```
def fonction(**kwargs):
    print(kwargs)

fonction(arg1=1, arg2=2) # affiche "{'arg1': 1, 'arg2': 2}"
```

Arguments de base

Ce dernier d'argument est à la fois le plus simple et le plus courant, et il supporte les deux types précédent. Cet argument peut donc être appelé par son nom (et l'ordre n'a pas d'importance) ou alors sans le nom, et dans ce cas-là, c'est leur position qui est déterminante.

```
def fonction(arg1, arg2, arg3):
    print(arg1, arg2, arg3)

fonction(1, arg3=3, arg2=2) # affiche "1 2 3"
fonction(1, 2, 3) # affiche "1 2 3"
```

Ainsi, une fonction qui utilise les trois types d'arguments pourrait s'écrire comme :

```
def fonction(positionnel_1, positionnel_2, /, arg1, arg2, *,
↪ kwarg1, kwarg2):
    pass
```

De la même manière une fonction qui pourrait prendre en compte n'importe quel nombre d'argument de n'importe quel type s'écrit :

```
def fonction(*args, **kwargs):
    pass
```

Cette méthode d'appel peut-être utile pour préciser des options dans une fonction (comme c'est le cas pour beaucoup de fonction de la bibliothèque `matplotlib`) ou exécuter une fonction dont on ne connaît a priori pas les arguments.

Argument par défaut

Il est possible de préciser une valeur par défault pour n'importe quel argument d'une fonction, en ajoutant `=valeur` dans la définition dans la fonction. La seule précaution à prendre est qu'un argument ayant une valeur par défaut ne peut pas précéder un argument sans valeur par défaut.

```
def fonction(arg1, arg2=2):
    print(arg1, arg2)

fonction(1) # affiche "1 2"
fonction(1, 3) # affiche "1 3"
```

Inversement, la syntaxe suivante est invalide :

```
def fonction(arg1=1, arg2):
    print(arg1, arg2)
```

Les arguments par défault sont cumulables avec les types d'arguments précédemment, il est ainsi possible d'avoir des arguments positonnel ou par mot-clef ayant une valeur par défaut.

2.1.3 La récursivité, c'est la récurvité, mais en plus simple

Définition de la récursivité

Une fonction est dite récursive si elle intervient dans sa propre définition. Le concept est dur à prendre en main car il n'est pas naturel de réfléchir en terme de récurvité. Par opposition, les fonction non récursives sont dites "itératives". Autrement dit, une fonction récursive va être de la forme :

```
def ma_fonction(<arguments>):
    <actions>
    ma_fonction(...)
    <action>
    return <variable>
```

Mise en place d'un algorithme récursif

La mise en place d'algorithme récursif va souvent s'axer sur deux grands principes :

— Soit une certaine condition est vraie et on renvoie à l'utilisateur le résultat. (on parle de cas terminal).

— Soit cette condition est fausse et l'on appelle la fonction récursive avec des arguments différents.

Cette mise en place est particulièrement simple sur des exemples mathématiques où l'on veut programmer une formule donnée par récurrence. Le principe est le même ici.

Mise en pratique

À titre d'exemple reprenons la construction de la fonction factorielle :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{sinon} \end{cases}$$

Ainsi, la fonction va naturellement s'articuler autour de la formule mathématique :

```
def factorielle(n):
    if n == 0:
        return 1
    else:
        return n * factorielle(n - 1)
```

2.2 Modules et importations

2.2.1 Principe des modules

Python connaît quelques fonctions indispensables, comme l'addition de deux nombres ou l'affichage d'un texte. Mais bien souvent, on a besoin de fonctions particulières, notamment des fonctions mathématiques, cos, sin, racine carrée, etc. Ces fonctions-là ne sont pas fournies dans Python directement, mais dans un "module" à part.

Un module est donc une sorte de grand script avec beaucoup de fonctions dedans. Nous verrons ici comment importer un module, et quelques modules usuels.

2.2.2 Importer un module

Il y a trois manière d'importer un module :

— `import module` : la manière la plus élégante et propre [1]

1. On peut également déclarer un alias : `import module as m`, la syntaxe pour les fonction devient alors `m.fonction()` au lieu de `module.fonction()`

— `from module import fn_1, fn_2, ...` : importe les fonction `fn_1` et `fn_2` du module.

— `from module import *` : importe toutes les fonctions du module. On connaît toutefois rarement toutes les fonctions d'un module, et il existe dès lors un risque qu'une fonction définie par nous entre en conflit avec une fonction du module. Cette syntaxe est donc à éviter et sera bannie de la suite de notre développement.

Revenons sur les deux premières méthode avec un exemple : (le module `math` ajoute des fonctions mathématiques)

```
>>> import math
>>> math.cos(0)
1.0
```

On aurait pu aussi écrire :

```
>>> from math import cos
>>> cos(0)
1.0
```

Ainsi, si la première syntaxe permet d'importer toutes les fonctions du module, la deuxième a l'avantage de dispenser le programmeur d'utiliser le préfixe `math.` chaque fois qu'il souhaite appeler la fonction.

2.2.3 Quelques modules usuels

Nous allons voir ici trois modules de manière succinte :

Le module `math` qui contient un grand nombre de fonctions mathématiques

Le module `random` qui permet de générer des nombres aléatoires

Le module `numpy` qui permet de manipuler des tableaux, ce module sera vu plus tard (voir 4.2)

Fonctions mathématiques

Il faut importer le module `math` : `import math` Donnons ici les principales fonctions :

`math.pi` une approximation de π

`math.cos(x)` fonction cosinus

`math.sin(x)` fonction sinus

`math.tan(x)` fonction tangente

`math.log(x)` fonction logarithme népérien

`math.sqrt(x)` racine carrée

Aléatoire

Le module correspondant s'appelle `random` : `import random`

`random.random()` renvoie un nombre aléatoire dans $[0\ ;\ 1[$

`random.randint(min, max)` renvoie un nombre entier entre $[min\ ;\ max]$

`random.choice(liste)` renvoie un élément au hasard de la liste (marche aussi avec une chaîne de caractère).

2.3 Applications

Les fonctions, sont, avec les bases vues au chapitre 1 (voir page 6), le cœur du langage. On peut donc d'ores et déjà faire quelques fonctions :

2.3.1 Théorème de pythagore

(Corrigé : 8.1.1)

Le but est de faire une petite fonction qui prend comme argument trois entiers (les trois côtés du triangles) qui renvoie un booléen qui vaut `True` si le triangle est rectangle, `False` sinon. On suppose que le côté le plus long est toujours donné en dernier.

2.3.2 Implémentation de la fonction factorielle

(Corrigé : 8.1.2)

La fonction factorielle est définie par :

$$n! = \prod_{i=1}^{n} i \quad \text{et} \quad 0! = 1$$

Le but est d'écire une fonction factorielle qui prend en argument n et renvoie $n!$.

2.3.3 Test de la primalité d'un nombre

(Corrigé : 8.1.3)

Un nombre $p \in \mathbb{N}^*$ est premier s'il n'est divisible par aucun entier $n \in [\![2\ ;\ \sqrt{p}]\!]$

Le programme va donc s'axer sur :

1. Pour tout les $n \in [\![2\ ;\ \sqrt{p}]\!]$
2. Si p est divisible par n
3. Alors on quitte la fonction : le nombre n'est pas premier
4. Si on a testé tous les nombres, alors p n'est divisible par aucun entier dans $[\![2\ ;\ \sqrt{p}]\!]$, donc p est premier

2.3.4 Version récursive de l'algorithme d'Euclide

(Corrigé : 8.1.4)

L'algorithme d'Euclide (une version itérative est proposé au : 6.2) permet de calculer le plus grand dénominateur commun entre deux nombres. Le but est d'en écrire une version récursive. Le programme devra calculer $PGCD(a\ ;\ b)$. Quelques jalons pour guider l'exercice :

1. tant que b est non nul
2. a prend b pour valeur et b prend $a\ MOD\ b$ pour valeur (c'est simultané, avec une notation indicielle où l'indice est le numéro du tour de boucle nous aurions : $a_{i+1} = a_i$ et $b_{i+1} = a_i\ MOD\ b_i$)

Chapitre 3

Chaînes de caractères, listes et dictionnaires

3.1 Chaînes de caractères

3.1.1 Introduction aux chaînes de caractères

Une chaîne de caractères est une suite ordonnée de caractères (lettres, nombres, etc.). À noter qu'il peut s'agir également d'une chaîne vide (qui ne contient aucun caractère) ou d'une chaîne à un seul caractère. La seule chose importante est de bien mettre la chaîne entre guillemets lors de l'initialisation.

3.1.2 La concaténation

Pour concaténer ("coller bout à bout") deux chaînes de caractères, il suffit de les "additionner" :

```
>>> ma_chaine1 = "Bonjour "
>>> ma_chaine2 = "tout le monde !"
>>> ma_chaine1 + ma_chaine2
'Bonjour tout le monde !'
```

3.1.3 La répétition

(Le terme n'est pas officiel) L'opérateur $*$ sert à multiplier une chaîne de caractères par un nombre entier p. On obtient la chaîne donnée en entrée concaténée p fois avec elle-même :

```
>>> "abc" * 3
'abcabcabc'
>>> ma_chaine = "Une chaîne "
>>> ma_chaine * 2
'Une chaîne Une chaîne'
```

3.1.4 Longueur d'une chaîne de caractères

On utilise la fonction `len` : `len(ma_chaine)` renvoie le nombre de lettres de la chaîne.

```
>>> ma_chaine = "Bonjour"
>>> len(ma_chaine)
7
```

3.1.5 Accéder à un caractère de la chaîne

Une chaîne de caractères peut être schématisée par un tableau. Chaque case du tableau vérifie :

— chaque case contient une lettre.

— chaque case est repérée par deux indices, l'un positif partant du début de la chaine et l'autre strictement négatif partant de la fin de la chaine

Prenons l'exemple de la chaîne `"Bonjour"` :

B	o	n	j	o	u	r
0	1	2	3	4	5	6
-7	-6	-5	-4	-3	-2	-1

On accède à un caractère de la chaine en écrivant : `ma_chaine[indice]` avec `indice`, un indice du caractère de la chaîne.

```
>>> ma_chaine = "Bonjour"
>>> ma_chaine[0]
'B'
>>> ma_chaine[3]
'j'
>>> ma_chaine[-7]
'B'
>>> ma_chaine[-4]
'j'
```

3.1.6 Extraction d'une sous-chaînes de caractères

Une manipulation un peu plus délicate sur les chaînes de caractères est d'extraire une sous-chaîne. On utilise alors la syntaxe `ma_chaine[debut: fin]` avec `debut` l'indice du premier caractère à prendre et `fin` l'indice du dernier caractère (ce caractère ne sera pas compris dans la sous-chaîne extraite).

Il existe des variantes de cette syntaxe selon l'usage voulu :

— `ma_chaine[:fin]` pour prendre du début de la chaîne jusqu'à `fin` (syntaxe équivalente de `ma_chaine[0: fin]`).

— `ma_chaine[debut:]` pour prendre à partir de `debut` jusqu'au dernier caractère (inclus).

Quelques exemples :

```
>>> ma_chaine = "Ceci est une phrase"
>>> ma_chaine[:4]
'Ceci'
>>> ma_chaine[5: 8]
'est'
```

```
>>> ma_chaine[-6:]
'phrase'
```

3.1.7 Appartenance d'une chaîne à une autre

Pour tester si une chaîne de caractères est incluse dans une autre, on utilise le mot-clef `in` : `chaine_1 in chaine_2`. Cette syntaxe renvoie `True` si `chaine_1` est incluse dans `chaine_2`.

Quelques exemples :

```
>>> ma_chaine = "Ceci est une phrase"
>>> "u" in ma_chaine
True
>>> "est" in ma_chaine
True
>>> "bonjour" in ma_chaine
False
```

3.1.8 Caractères spéciaux

Il est parfois nécessaire de stocker dans une chaîne de caractères des caractères spéciaux. Ces caractères peuvent être, entre autres, un retour à la ligne, une tabulation, des guillemets etc.

Tout ces caractères spéciaux sont introduit par la syntaxe : \<`caractère`>.

Syntaxe	Effet
\n	retour à la ligne
\t	tabulation
\"	guillemet

Il en existe d'autre, mais l'intérêt étant plutôt limité, ne sont présenté ici que les plus courants.

Par exemple :

```
>>> ma_chaine = "Ceci est sur une ligne\nEt ça sur une autre\nOn
 ↪  peut du texte entre \"guillemets\""
>>> print(ma_chaine)
Ceci est sur une ligne
Et ça sur une autre
On peut mettre du texte entre "guillemets"
```

3.2 Listes

3.2.1 Introduction aux listes

Une liste est une... liste de variables. On parle de type composé. Quelques points importants sur les listes :

- Pour désigner une variable de la liste, on parle d'élément de la liste
- On accède à un élément de la liste via la syntaxe : `ma_liste[indice]` avec `indice` un indice de l'élément dans la liste.
- Les indices sont doubles, exactement comme pour les chaînes de caractères (voir 3.1)

Les listes ont beaucoup de points communs avec les chaînes de caractères, notamment au niveau des indices et de l'extraction de sous-listes. Ce type possède néanmoins de nombreuses autres fonctionnalités.

3.2.2 Initialiser une liste

Contrairement aux entiers ou aux chaînes de caractères il y a plusieurs méthodes pour initialiser une liste. La première et la plus intuitive est de placer les éléments de la liste entre crochets (on dit alors que la liste est définie en extension) :

```
>>> ma_liste = [1, 2, 3]
>>> ma_liste
[1, 2, 3]
```

Une autre méthode d'initialisation est plus délicate et nécessite d'avoir vu les boucles itératives `for` (voir 1.4). L'idée est de remplir une liste grâce à une boucle itérative, on parle alors de liste définie en compréhension. La syntaxe est : `ma_liste = [val for i in range(n)]`, ce qui crée une liste de longueur `n`, et chaque élément de la liste vaut `val`. On peut imaginer plusieurs cas de figures :

```
>>> ma_liste = [0 for _ in range(4)]
>>> ma_liste
[0, 0, 0, 0]
>>> ma_liste = [i for i in range(5)]
>>> ma_liste
[0, 1, 2, 3, 4]
>>> ma_liste = [i ** 2 for i in range(1, 6)]
>>> ma_liste
[1, 4, 9, 16, 25]
```

3.2.3 Accéder à l'élément d'une liste

Le premier élément de la liste est à l'indice 0, et on appelle l'élément n de la liste grâce à la syntaxe : `ma_liste[n]`.

```
>>> ma_liste = [1, 2, 3]
>>> ma_liste[0]
1
>>> ma_liste[1]
2
>>> ma_liste[3] # Si on sort de la liste, Python renvoie une
 ↪  erreur
IndexError: list index out of range
```

Comme pour les chaînes de caractères, on peut accéder à un élément en partant de la fin :

```
>>> ma_liste = [1, 2, 3]
>>> ma_liste[-1]
3
```

3.2.4 Longueur d'une liste

C'est exactement la même chose que pour les chaînes de caractères (voir 3.1.4). La fonction renvoie alors le nombre d'éléments de la liste.

```
>>> ma_liste = [1, 2, 3]
>>> len(ma_liste)
3
```

3.2.5 Extraire une sous-liste

Comme pour les chaînes de caractères, on peut extraire une sous-liste de la liste. La syntaxe et le principe sont les mêmes. Donnons quelques exemples :

```
>>> ma_liste = [1, 2, 3, 4, 5]
>>> ma_liste[:3]
[1, 2, 3]
>>> ma_liste[-3: -2]
[3]
```

3.2.6 Concaténer deux listes

Comme pour les chaînes de caractères, on utilise l'opérateur + :

```
>>> ma_liste1 = [1, 2, 3]
>>> ma_liste2 = [4, 5, 6]
>>> ma_liste1 + ma_liste2
[1, 2, 3, 4, 5, 6]
```

3.2.7 Répétition d'une liste

L'idée est la même que pour les chaînes de caractères : la liste va être dupliquée (concaténée à elle-même autant de fois qu'il est indiqué).

```
>>> ma_liste = [1, 2]
>>> ma_liste * 3
[1, 2, 1, 2, 1, 2]
>>> [0] * 5
[0, 0, 0, 0, 0]
>>> [0] * 3 + [1, 2] * 2
[0, 0, 0, 1, 2, 1, 2]
```

3.2.8 Chercher un élément

Pour savoir si un élément est dans une liste, on utilise le mot-clef `in`, dans la syntaxe : `element in ma_liste` avec `element` l'élément recherché.

Cette syntaxe renvoie un booléen (à savoir `True` ou `False`) on peut donc s'en servir comme d'une condition. Par exemple, dans un petit script :

```python
# On calcule les carrés de 1 à 50
carres = [i ** 2 for i in range(1, 51)]

if 49 in carres:
    print("49 est dans la liste")
else:
    print("49 n'est pas dans la liste")
```

3.2.9 Ajouter un élément à la fin d'une liste

La syntaxe n'est pas évidente : `ma_liste.append(element)` avec `ma_liste` une liste et `element` l'élément à ajouter à la fin.

```
>>> ma_liste = [1, 2, 3]
>>> ma_liste
[1, 2, 3]
>>> ma_liste.append(4)
>>> ma_liste
[1, 2, 3, 4]
>>> ma_liste.append("Coucou")
>>> ma_liste
[1, 2, 3, 4, "Coucou"]
```

3.2.10 Enlever un élément

Par valeur

On peut enlever un élément d'une liste selon sa valeur. La syntaxe est alors `ma_liste.remove(valeur)` avec `valeur`, la valeur de l'élément que l'on veut enlever. À noter que cela ne supprime que la première occurrence de la valeur.

```
>>> ma_liste = [0] * 3 + [1, 2] * 2
>>> ma_liste
[0, 0, 0, 1, 2, 1, 2]
>>> ma_liste.remove(0)
>>> ma_liste
[0, 0, 1, 2, 1, 2]
>>> ma_liste.remove(2)
>>> ma_liste
[0, 0, 1, 1, 2]
```

Par indice

On cherche à supprimer un élément d'indice donné. On a la syntaxe : `ma_liste.pop(indice)`, avec `indice` un indice de l'élément à supprimer.

```
>>> ma_liste = [0] * 3 + [1, 2] * 2
>>> ma_liste
```

```
[0, 0, 0, 1, 2, 1, 2]
>>> ma_liste.pop(0)
0
>>> ma_liste
[0, 0, 1, 2, 1, 2]
>>> ma_liste.pop(2)
1
>>> ma_liste
[0, 0, 2, 1, 2]
```

Il peut être utile de savoir que `.pop`, contrairement à `.remove`, renvoie la valeur de l'élément supprimé.

3.3 Dictionnaires

3.3.1 Introductions aux dictionnaires

Un dictionnaire est un ensemble de couples. Le premier élément de chaque couple est appelé "clef" et le second est appelé "valeur". Le principe du dictionnaire consiste à permettre de retrouver la valeur sur la donnée de la clef (i.e. si on connaît la clef, on peut retrouver la valeur.). Attention, la réciproque est fausse, si on connait une valeur, rien ne garantit que nous allons pouvoir remonter jusqu'à la clef.

On peut faire une analogie avec les dictionnaires au sens usuel du terme : si on connaît l'orthographe d'un mot (la clef) on peut trouver sa définition (la valeur). Mais si nous n'avons que la définition, retrouver le mot en fouillant dans le dictionnaire risque d'être long...

3.3.2 Définir un dictionnaire

Il existe plusieurs manières de définir un dictionnaire.

La plus courante est sans doute de commencer par définir un dictionnaire vide, puis de le remplir à l'aide d'une boucle. On écrit alors : `mon_dico = {}`.

On peut également déclarer un dictionnaire avec la donnée d'un ensemble de couple clef-valeur : `mon_dico = {clef_1: valeur_1, clef_2: valeur_2, ...}`

Par si on veut prendre pour clef le nom d'une personne et pour valeur son adresse mail :

```
>>> mon_dico = {"Emmanuel Macron": "manu@gouv.fr", "Jean Michel
↪  Blanquer": "jean_michou@gouv.fr"}
>>> mon_dico = {} # un dictionnaire vide
>>> mon_dico = {1: [1, 2, 3], "b": 12} # autre exemple
```

3.3.3 Longueur d'un dictionnaire

Comme pour les chaînes et les listes (voir 3.1.4). La fonction `len` renvoie alors le nombre de couple du dictionnaire :

```
>>> mon_dico = {"a": [1, 2, 3], "b": [4, 5, 6]}
>>> len(mon_dico)
2
```

3.3.4 Ajouter un couple clef-valeur à un dictionnaire

La syntaxe est assez simple : `mon_dico[nouvelle_clef] = valeur`, avec `nouvelle_clef` la clef à ajouter et `valeur` la valeur correspondante.

```
>>> mails = {}
>>> mails["Manu"] = "manu@exemple.fr"
>>> mails
{'Manu': 'manu@exemple.fr'}
```

3.3.5 Lire la valeur à partir d'une clef

La syntaxe est proche de celle vu juste au-dessus : `mon_dico[clef]`, avec `clef` la clef qui correspond à la valeur que l'on souhaite avoir. Cette syntaxe retourne la valeur.

```
>>> mails = {"Manu": "manu@exemple.fr"}
>>> mails["Seb"] = "sebastien@osef.com" # on ajoute l'adresse
 ↪  mail de Seb
>>> mails["Manu"]
'manu@exemple.fr',
>>> mails["Seb"]
'sebastien@osef.com'
```

3.3.6 Savoir si une clef est enregistrée dans un dictionnaire

Pour savoir si une clef est dans un dictionnaire, il faut utiliser la syntaxe : `clef in mon_dico`. Comme pour les chaînes de caractères et les listes, cela renvoie un booléen.

Attention, il s'agit bien d'un test sur les clefs, et non les valeurs.

```
>>> mon_dico = {"a": [1, 2, 3], "b": [4, 5, 6]}
>>> "b" in mon_dico
True
>>> [1, 2, 3] in mon_dico
False
```

3.3.7 Retirer un couple clef-valeur d'un dictionnaire

La syntaxe est proche de celle sur les listes : `mon_dico.pop(clef)`, avec `clef`, la clef du couple à supprimer. Comme sur les listes, cette syntaxe renvoie la valeur supprimée.

```
>>> mon_dico = {1: "a", 2: "b", 3: "c"}
>>> mon_dico.pop(1)
'a'
>>> mon_dico
{2: 'b', 3, 'c'}
>>> mon_dico.pop("c")
KeyError: 'c'
```

Chapitre 4

Introduction aux outils pour l'analyse numérique

Ce chapitre a pour but d'introduire quelques outils utiles pour l'analyse numérique (calcul d'intégrale et solution approchée d'une équation différentielle).

4.1 Module `matplotlib.pyplot`

4.1.1 Intérêt du module

Ce module permet d'afficher des dessins à l'écran au travers de diverses fonctions. On peut ainsi afficher des nuages de points de différentes couleurs ou formes, afficher des courbes ou des graphes etc.

`matplotlib.pyplot` a la particularité de ne pas dessiner immédiatement à l'écran. C'est-à-dire que lorsqu'on demande l'affichage d'une figure (nuage de points, courbes etc) le programme n'affiche rien à l'écran : les figures à dessiner sont stockées en mémoire et sont affichées sur demande (i.e. grâce à une autre fonction).

Le nom du module étant particulièrement long, il est préférable de l'importer et de déclarer un alias : `import matplotlib.pyplot as plt`

4.1.2 Fonctions usuelles

Deux fonctions sont indispensables :

`matplotlib.pyplot.show()` affiche les figures stockées en mémoire

`matplotlib.pyplot.plot(X, Y)` stocke en mémoire la courbe reliant les points de coordonnées X_i ; Y_i. (`X` et `Y` sont des listes ou des tableaux numpy)

La fonction `matplotlib.pyplot.plot()` peut prendre bien d'autres arguments comme la couleur ou le style de points. Relativement peu utilisées, ces options ne sont pas à connaître.

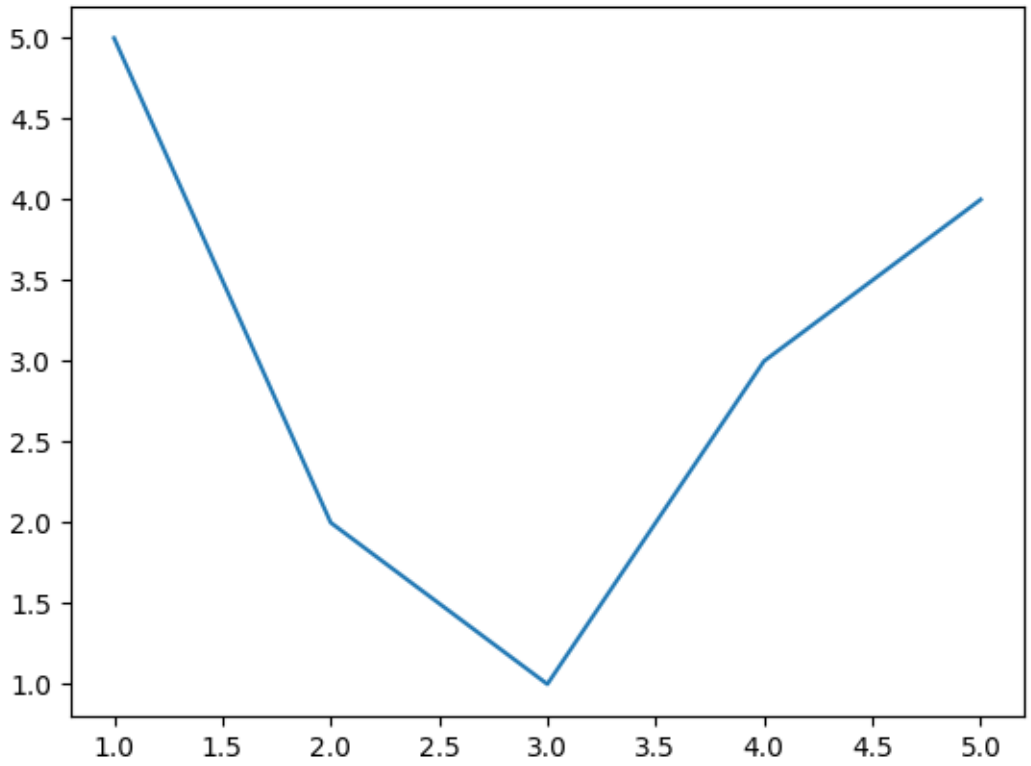
On peut également mentionner la fonction moins utilisée, mais néanmoins pratique : `matplotlib.pyplot.scatter(X, Y)`. Elle permet de tracer un nuage de points sans les relier.

4.1.3 Exemples

Un petit script qui trace une courbe :

```
1 import matplotlib.pyplot as plt
2
```

```
# Coordonnées des points
X = [1, 2, 3, 4, 5]
Y = [5, 2, 1, 3, 4]

# Création de la courbe reliant les points
plt.plot(X, Y)

# Affichage de la courbe
plt.show()
```



4.2 Module `numpy`

4.2.1 Introduction au module

Le module `numpy` fourni non pas des fonctions, mais un nouveau type de variable : le type `array`[1]. Ce type permet de construire des listes ou des tableaux de variables. Mais, contrairement aux listes Python, les `array` doivent respecter quelques règles :

— la dimension est fixée lors de l'initialisation et ne peut pas être modifiée ;

— le type `array` ne peut contenir que des variables d'un seul type (e.g. on peut faire des `array` de `int`, de `str`, d'`array`, etc.).

1. Aussi parfois appelé "Tableau `numpy`"

En contrepartie de ces restrictions, le type `array` permet des manipulations plus agréables et intuitives.

La dimension d'un tableau `numpy` est un tuple que l'on peut récupérer par la fonction : `numpy.shape(mon_array)` [2]. Si ce tuple ne contient qu'un seul nombre n, le tableau est unidimensionnel i.e. c'est une liste de longueur n ; s'il contient deux nombres m et n, alors le tableau est bidimensionnel i.e. c'est une grille dotée de m lignes et n colonnes ; si la dimension est constituée de trois nombres, le tableau est dit tridimensionnel, et les trois nombres désignent sa largeur, sa hauteur et sa profondeur ; et ainsi de suite.

On se contentera généralement de tableaux uni- ou bi-dimensionnels.

4.2.2 Tableaux `numpy`

Initialisation

Il existe plusieurs méthodes d'initialisations.

En énumérant les éléments Les tableaux `numpy` se comportent alors comme des listes Python, il faut donner les valeurs : `numpy.array(data)`. Où `data` est une liste Python (ou un tuple). Pour un tableau bidimensionnel, `data` est une liste de listes de même longueur qui représentent chacune une ligne du tableau. Pour un tableau tridimensionnel, `data` est une liste de listes de listes, et ainsi de suite.

Avec la dimension Si l'on connait la dimension exacte du tableau que l'on veut créer, on peut demander à `numpy` de construire un `array` de cette taille. Il faut alors préciser si l'on souhaite :

- un tableau de 0 : `numpy.zeros(dimension)`, avec `dimension` la dimension du tableau souhaité, sous forme de tuple ou de liste.
- un tableau de nombre aléatoires : `numpy.random.randint(minimum, maximum, dimension)`.

Pour une courbe Lorsque l'on veut calculer les points d'une courbe, on a souvent l'intervalle sur lequel il faut calculer les points et le pas [3] On utilise alors la fonction : `numpy.arange(debut, fin, pas)`. À noter que comme pour le `range` du Python, la borne haute n'est pas atteinte.

Dans certains cas, il est préférable de laisser `numpy` choisir le pas. Il faut alors lui donner un autre argument : le nombre de points voulu. Il faut utiliser une autre fonction : `numpy.linspace(debut, fin, nombre_points)`. Attention, ici la borne supérieure est atteinte.

Effets des opérateurs et fonctions

Les opérateurs sur les tableaux `numpy` ont une signification différentes de celle qu'ils ont sur les listes Python.

En effet, sur les `array`, les opérateurs correspondent à des opérations terme à terme sur les éléments du tableau. Ainsi, additionner deux tableaux `numpy` revient à sommer terme à terme les éléments des tableaux. Pour avoir le droit d'effectuer une opération sur deux tableaux `numpy`, il faut donc que les deux tableaux soient de même dimension. On peut également additionner avec un scalaire pour ajouter la même valeur à toutes les cases. De même pour la soustraction, la multiplication, la division et la division euclidienne (quotient et reste).

2. On pourrait également utiliser la syntaxe : `mon_array.shape`.
3. L'écart entre chaque points. Plus l'écart est grand, moins il y aura de points.

On peut également appliquer des fonctions mathématiques sur l'ensemble d'un tableau `numpy`. La syntaxe est alors plus complexe : `numpy.vectorize(fonction)(tableau)`, où `fonction` est la fonction mathématique à appliquer sur `tableau`.

Extraction d'un sous-tableau

Comme pour les listes (voir 3.2.5), on peut extraire un sous-tableau d'un tableau `numpy`, et on va là encore utiliser la syntaxe : `tableau[debut: fin]`. La subtilité est qu'un tableau peut être n-dimensionnel. Il va donc falloir indiquer la zone à garder, et ce pour chaque dimension.

La syntaxe est alors : `tableau[debut_1: fin_1, debut_2: fin_2, ..., debut_n: fin_n]`. Comme pour les listes, l'indice de fin n'est jamais atteint et l'on peut utiliser des indices négatifs pour compter à partir de la fin.

4.2.3 Exemples

Initialisation

```
>>> import numpy as np
>>> np.array([[1, 2], [3, 4]])
array([[1, 2],
       [3, 4]])
>>> np.zeros((3))
array([0., 0., 0.])
>>> np.arange(0, 5, 0.5)
array([0., 0.5, 1., 1.5, 2., 2.5, 3., 3.5, 4., 4.5])
```

Manipulations élémentaires

```
>>> from math import cos
>>> import numpy as np
>>> tableau = np.array([[1, 2], [3, 4]])
>>> tableau + 1
array([[2, 3],
       [4, 5]])
>>> np.vectorize(cos)(tableau)
array([[ 0.54030231, -0.41614684],
       [-0.9899925 , -0.65364362]])
```

Extraction d'un sous-tableau

```
>>> import numpy as np
>>> tableau = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> tableau
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> tableau[:, 0] # On ne garde que le premier élément de chaque
 ↪  ligne
array([1, 4, 7])
```

```
>>> tableau[1, 0]
4
>>> tableau[1:, 1:] # On garde tout à partir de la deuxième
 ↪  colonne et de la deuxième ligne
array([[5, 6],
       [8, 9]])
>>> tableau[:2, 1:2] # On ne garde que les deux premières lignes
 ↪  de la deuxième colonne
array([[2],
       [5]])
>>> tableau[:-1, 1:]
array([[2, 3],
       [5, 6]])
>>> tableau[1:-1, 0]
array([4])
```

4.3 Application

4.3.1 Calcul des points d'une fonction et affichage

(Corrigé : 8.2.1)

À l'aide de deux tableaux `numpy`, calculez et tracez la fonction $f : x \mapsto \dfrac{\sin(x)}{x}$. L'intervalle et le nombre de point sont laissés libre. Quelques grandes étapes :

1. Choisir un intervalle sur lequel f est définie et fixer un nombre de points.
2. Commencer par créer un tableau pour les abscisses
3. Construire le tableau pour les ordonnées
4. Afficher le resultat

Chapitre 5

Gestion des fichiers

Ce chapitre, plutôt court, a pour but de présenter les manipulations que l'on peut faire sur des fichiers extérieurs avec Python.

5.1 Ouvrir un fichier

5.1.1 Chemins relatif et absolu

Pour ouvrir, lire ou écrire dans un fichier, il faut d'abord savoir où ce fichier est stocké dans la mémoire de l'ordinateur. Chaque fichier peut être trouvé dans un dossier lui-même contenu dans un dossier, et ainsi de suite, jusqu'à ce qu'on atteigne le dossier qui contient tous les autres, qu'on appelle la "racine". La suite de dossiers qui mène de cette racine à un fichier est appelée "chemin" ou "arborescence".

Chemin absolu

Le chemin absolu permet à l'ordinateur de retrouver le fichier voulu indépendemment de l'emplacement du script Python. Il donne en fait la totalité de l'arborescence du fichier depuis la racine.

Cette méthode est pratique au sens où l'on peut changer de place le script Python sans que cela affecte son fonctionnement. Mais elle a aussi l'inconvénient de ne marcher que sur l'ordinateur sur lequel le script a été programmé. En effet le chemin absolu contient des informations propres à chaque ordinateur (notamment le nom d'utilisateur, ou le nom d'un dossier spécifique).

Par exemple, on peut imaginer une arborescence comme celle-ci :

```
/home/utilisateur/Bureau/fichier.txt
```

Le caractère `/` au début indique que l'on part de la racine de l'ordinateur, puis dans le dossier `home`, puis `utilisateur`, `Bureau` et enfin, on s'intéresse au fichier `fichier.txt`

Chemin relatif

Le chemin relatif ne part pas de la racine, mais de l'emplacement du script Python. Cette méthode a l'avantage d'être adaptée au transfert d'une machine à une autre. Mais le simple fait de changer le script Python de place suffit à l'empêcher de fonctionner.

Par exemple, nous allons créer un dossier `python` sur le Bureau. Dans ce dossier, nous allons placer un script Python d'un nom quelconque ainsi qu'un dossier `fichiers` qui contiendra les fichiers dont le script aura besoin. Pour l'exemple, on prendra deux fichiers : `fichier_1.txt` et `fichier_2.txt`

Si, depuis le script Python, on veut accéder à `fichier_1.txt`, il suffira de prendre le chemin relatif :

```
fichiers/fichier_1.txt
```

(Un chemin relatif ne commence jamais par le symbole : `/`.)
Ainsi, lors de l'utilisation de chemins relatifs, il est commode de créer un dossier avec le script Python et l'ensemble des fichiers qui sont susceptibles d'être appelés. Ainsi, changer le dossier de place n'affectera pas le fonctionnement du script.

5.1.2 Ouvrir un fichier

Pour ouvrir un fichier, nous allons faire appel à la fonction `open` qui va prendre deux arguments :

le chemin absolu ou relatif du fichier à ouvrir ;

le mode d'ouverture un paramètre qui régit les manipulations que l'on veut effectuer sur le fichier (nous verrons ces modes plus en détatil par la suite.).

On a donc une syntaxe de ce type : `mon_fichier = open("chemin/fichier_1.txt", "mode")`. Cette fonction renvoie une variable qui correspond au fichier et sur laquelle on peut effectuer des opérations (généralement lire ou écrire). Ces opérations devront néanmoins respecter le mode d'ouverture précisé lors de l'appel à la fonction.

Lorsque l'on a terminé de manipuler le fichier, il est important de le fermer via la syntaxe : `mon_fichier.close()`

5.2 Lire un fichier

Il faut preciser `"r"` (pour "read") en mode d'ouverture du fichier, la syntaxe devient : `mon_fichier = open("chemin/fichier_1.txt", "r")`.

On accède ensuite au contenu lui-même via la commande : `contenu = mon_fichier.read()`. Il existe deux autres fonctions qui permettent de lire dans un fichier :

— `mon_fichier.readline()` qui renvoie la ligne suivante du document. (Lors du premier appel, la fonction renvoie la première ligne, puis la deuxième et ainsi de suite. Lorsqu'il n'y a plus de ligne, la fonction renvoie la chaîne vide.)

— `mon_fichier.readlines()` qui renvoie une liste qui contient toutes les lignes du document.

Et on oublie pas de fermer le fichier après. La fermeture du fichier ne modifie pas la variable `contenu`. On peut donc tout à fait fermer le fichier juste après avoir sauvegardé le contenu.

5.3 Écrire dans un fichier

Il faut distinguer deux types d'écriture :

— On ajoute du contenu à la fin du fichier, le mode d'ouverture est alors `"a"`, pour "append".

— On supprime le contenu existant et on écrit le contenu voulu dans le fichier, le mode d'ouverture est : `"w"` pour "write".

Dans les deux cas, il suffit ensuite d'utiliser la commande : `mon_fichier.write("...")` pour écrire dans le fichier.

Notez qu'écrire dans un fichier qui n'existe pas créé le fichier.

5.4 Récapitulatif

Pour manipuler des fichiers, on utilise la fonction `open`. Cette fonction prend en argument :

1. le chemin (relatif ou absolu) du fichier à ouvrir ;
2. le mode d'ouverture.

Mode d'ouverture	Utilisation
`"r"`	lire le contenu d'un fichier
`"w"`	écraser le contenu d'un fichier avec un nouveau contenu
`"a"`	ajouter le nouveau contenu à la fin d'un fichier

5.5 Mise en pratique

5.5.1 Organisation du dossier

Pour l'exemple, nous reprendrons la configuration prise lors de l'introduction des chemins relatifs (voir 5.1.1). Nous allons donc prendre un dossier principal (son nom importe peu) qui va contenir :

1. un script python (`test_fichier.py` dans la suite) ;
2. un dossier du nom de `fichiers` qui ne contient rien pour l'instant.

5.5.2 Écriture puis lecture d'un fichier

Le but ici est de créer un fichier contenant un texte à l'intérieur du dossier. Nous allons donc utiliser la commande `open` avec ces arguments :

1. le chemin : `fichiers/mon_fichier.txt`
2. le mode d'ouverture : ici, une ouverture en écriture, donc `"a"` ou `"w"` (cela n'a pas d'importance, étant donné que le fichier n'existe pas, le contenu ne sera pas écrasé).

Le code dans le fichier python va donc être :

```
1 mon_fichier = open("fichiers/mon_fichier.txt", "a")
2 mon_fichier.write("Ceci est une phrase.")
3 mon_fichier.close()
```

Vous pouvez ensuite ouvrir le fichier nouvellement créé avec un éditeur de texte normal pour vous assurer que l'écriture s'est bien passée.

On peut également lire le contenu du fichier (attention, pour lire, il faut que le fichier existe). Le code devient alors :

```
mon_fichier = open("fichiers/mon_fichier.txt", "r")
contenu = mon_fichier.read()
mon_fichier.close()

print(contenu)
```

Ce qui suit n'est pas follement intéressant c'est juste pour montrer l'utilisation de `readlines`.

```
mon_fichier = open("fichiers/mon_fichier.txt", "a")
mon_fichier.write("Ceci est une phrase\nEt encore une parce que sinon c'est...\npas intéressant.")
mon_fichier.close()

mon_fichier = open("fichiers/mon_fichier.txt", "r")
print(mon_fichier.readlines())
```

On pourrait ensuite analyser le contenu du fichier en utilisant les manipulation vues sur les chaînes de caractères (voir 3.1), la variable `contenu` est en effet de type `str`.

Chapitre 6

Algorithmes usuels et notions théoriques

6.1 Notions préliminaires

6.1.1 Invariant de boucle

On appelle "invariant de boucle" une assertion mathématique qui est vraie à chaque tour de boucle. Lors de l'exécution d'un algorithme, on peut ainsi suivre son avancement via l'invariant de boucle.

Cet invariant peut servir à démontrer la correction[1] d'un algorithme en passant par une récurrence.

6.1.2 Complexité d'un algorithme

La complexité est une grandeur qui caractérise le nombre de calculs que l'algorithme demande. L'avantage de cette grandeur est qu'elle est propre à l'algorithme et n'est donc pas influencée par la machine qui exécute le programme.

Pour déterminer la complexité, on compte le nombre d'opérations d'un certain type, choisi de manière à avoir une grandeur pertinente. On peut, par exemple, choisir de compter les comparaisons, les multiplications, les affectations de variables etc. La complexité doit s'exprimer en fonction de la taille des données d'entrées (la longueur d'une liste, le nombre de cases d'un tableau, le nombre de chiffres d'un nombre, etc.).

6.1.3 Terminaison d'un algorithme

Vérifier la terminaison d'un algorithme, c'est vérifier qu'il ne tournera pas indéfiniment.

C'est généralement assez simple à prouver. On peut décomposer les preuves de terminaison d'algorithme en trois cas :

— Les `for` : le cas le plus simple, il y a un nombre fini de tour de boucle, il reste à s'assurer qu'aucune opération illicite n'est effectuée dans la boucle (pas de division par zéro, pas d'accès à un élément d'une liste qui n'existe pas ou qui est en dehors de la liste...)

— Les `while` : il faut s'assurer que la condition est fausse au bout d'un nombre fini de tour, cela peut s'avérer délicat, et l'on aura souvent recourt au point suivant. Il faut également vérifier qu'aucune opération illicite n'est effectuée.

— Le principe de descente infinie de Fermat : toute suite d'entier positive et strictement décroissante atteint 0 au bout d'un temps fini.

1. la correction d'un algorithme est une preuve mathématique du fait que l'algorithme fait bien ce que l'on veut.

6.2 Euclide : Calcul du plus grand diviseur commun

6.2.1 Principe

L'algorithme d'Euclide permet de calculer le plus grand diviseur commun entre deux entiers. Le principe repose sur la propriété du PGCD : $PGCD(a\ ;\ b) = PGCD(b\ ;\ r)$ en notant r le reste de la division euclidienne de a par b, de plus $PGCD(a\ ;\ 0) = a$. Cette propriété du PGCD consistue l'invariant de boucle.

L'invariant de boucle est ici : $PGCD(a\ ;\ b) = PGCD(b\ ;\ r)$

6.2.2 Terminaison

En notant a_n et b_n les valeurs de a et b au $n^{\text{ième}}$ tour. Au début, nous avons : $b_0 = b$. Et à chaque tour, $b_{n+1} = a_n\ MOD\ b_n$. Or, par définition du reste de la division euclidienne, $a\ MOD\ b < b$.

Ainsi $(b_n)_{n\in\mathbb{N}}$ est une suite positive et décroissante d'entiers, donc, d'après le principe de descente infinie de Fermat :

$$\exists N \in \mathbb{N},\ b_N = 0$$

D'où la terminaison de l'algorithme.

6.2.3 Algorithme

```
1  def pgcd(a, b):
2      # tant que b est non nul
3      while b != 0:
4          # a prend la valeur b
5          # b prend pour valeur le reste de la division euclidienne
           ↪  de a par b
6          a, b = b, a % b
7
8      # b est nul, PDCG(a, 0) = a, on renvoie donc a
9      return a
```

6.3 Bézout : résolution d'équations diophantiennes

6.3.1 Principe

Le principe de l'algorithme de Bézout est de résoudre des équations de la forme : $au + bv = PGCD(a\ ;\ b)$. Aussi appelé "algorithme d'Euclide étendu", l'algorithme de Bézout calcule le plus grand diviseur commun à a et b, ainsi qu'une paire de coefficients $(u\ ;\ v) \in \mathbb{Z}^2$.

L'invariant de boucle est :

$$\begin{aligned} PGCD(r1\ ;\ r2) &= PGCD(a\ ;\ b) \\ r1 &= au + bv \\ r2 &= as + bt \end{aligned}$$

6.3.2 Terminaison

La terminaison est la même que pour l'algorithme d'Euclide, avec $a = r1$ et $b = r2$.

6.3.3 Algorithme

```python
def bezout(a, b):
    # on initialise les variables
    u, v, s, t = 1, 0, 0, 1
    r1, r2 = a, b

    # tant que b est non nul
    while r2 != 0:
        # q prend pour valeur le quotient de la division
        ↪ euclidienne de a par b
        q = r1 // r2
        r1, u, v = = r2, s, t
        r2 = r1 % r2
        s = u - q * s
        t = v - q * t

    # on retourne le PGCD ainsi que les coefficients (u; v)
    return r1, u, v
```

6.4 Dichotomie

Les algorithmes de dichotomies sont employés pour recherher un nombre dans une liste de nombres triés. Cette méthode à l'avantage d'avoir une très bonne complexité par rapport aux algorithmes de recherche qui parcourent la liste en entier.

6.4.1 Dans une liste triée

Principe

On dispose d'une liste triée `ma_liste`, et on sait que si l'élément `element` que l'on cherche est dans la liste, alors il est dans la sous-liste `ma_liste[mini: maxi + 1]`.

Au début on pose de l'algorithme : `mini = 0` et `maxi = len(ma_liste) - 1`. En effet, si l'élément est dans la liste, alors il est dans la sous-liste `ma_liste[mini: maxi + 1]` (qui correspond alors à la liste entière).

Au tour suivant, on calcule l'indice central de la liste : `milieu = (mini + maxi) // 2`. Or la liste est triée, donc :

— si `element > ma_liste[milieu]`, alors si l'élément est dans `ma_liste`, il est dans la sous-liste : `ma_liste[milieu + 1: maxi]`

— si `element <= ma_liste[milieu]`, alors si l'élément est dans `ma_liste`, il est dans la sous-liste : `ma_liste[mini: milieu]`

Cet algorithme a une complexité en $\Theta(\log n)$ où n est la longueur de la liste donnée en argument.

Terminaison

Et tant que la condition `mini > maxi` est vraie on tourne. À chaque tour, l'écart `maxi - mini`, non seulement diminue, mais est divisé par deux. Donc le programme va donc

nécessairement terminer (principe de descente inifinie de Fermat). Et à la fin, si l'élément est dans `ma_liste`, alors `element = ma_liste[mini]` (ou `element = ma_liste[maxi]`).

Algorithme

```
def dichotomie(ma_liste, element):
    # on initialize les variables
    mini = 0
    maxi = len(ma_liste) - 1

    while mini > maxi:
        # on calcule l'indice du milieu de la sous-liste
        milieu = (mini + maxi) // 2

        # on regarde dans quelle moitié de la sous-liste
        ↪ l'élément peut être
        if element > ma_liste[milieu]:
            mini = milieu + 1
        else:
            maxi = milieu

    return element == ma_liste[mini]
```

6.4.2 Sur une fonction monotone

Principe

De manière analogue à la précédente ; on cherche à trouver le point d'annulation d'une fonction monotone sur un intervalle $[mini\ ;\ maxi]$ avec une certaine précision.

Comme pour la première version, on va calculer le milieu de l'intervalle et distinguer deux cas :

— si $f(mini) \cdot f(milieu) > 0$: $f(mini)$ et $f(milieu)$ sont donc de même signe or la fonction est monotone, donc le point d'annulation est après le milieu de l'intervale

— sinon, le points d'annulation est avant le milieu de l'intervalle

En comptant le nombre de tour de boucle, la complexité est un $\Theta(n)$ où n est la précision (en bit) du résultat. Autrement dit, plus le résultat est précis, plus l'éxecution du programme sera longue.

Terminaison

En notant a_n et b_n les valeurs de a et b au $n^{\text{ième}}$ tour. Au sein d'un tour de boucle, on a :

$$\forall n \in \mathbb{N}^*,\ b_{n+1} - a_{n+1} = \frac{b_n - a_n}{2}$$

Par récurrence, au bout de k tours de boucle :

$$0 < b_k - a_k = \frac{b - a}{2^k}$$

Or, $\dfrac{b-a}{2^k} \underset{k\to+\infty}{\longrightarrow} 0$ ainsi :

$$\forall \varepsilon > 0,\ \exists N \in \mathbb{N}^*,\ |b_N - a_N| < 2\varepsilon$$

Algorithme

```
def dichotomie(f, mini, maxi, precision):
    while (maxi - mini) > 2 * precision:
        milieu = (maxi - mini) / 2
        if f(mini) * f(milieu) > 0:
            mini = milieu
        else:
            maxi = milieu
    return (maxi - mini) / 2
```

6.5 Méthode des rectangles et des trapèzes : approximation d'une intégrale

6.5.1 Principe

On considére une subdivision de l'intervalle sur lequel on veut approximer l'intégrale de f. Et pour chaque petit intervalle on va approximer la fonction par une fonction constante, ainsi, calculer l'aire sous la courbe va revenir à un calcul d'aire de rectangle (voir Figure 6.1).

On se place ainsi sur un segment $[a\ ;\ b]$ et on fixe un nombre arbitraire n de subdivisions sur ce segment. Les largeurs des subdivisions sont égales [2], et valent : $\dfrac{b-a}{n}$.

La fonction étant approximée par une fonction constante, la hauteur du $k^{\text{ième}}$ rectangle vaut $f(a + k * largeur)$. Ainsi l'aire sous le $k^{\text{ième}}$ rectangle est de : $\dfrac{b-a}{2} \cdot f\left(a + k \cdot \left(\dfrac{b-a}{2}\right)\right)$

2. On parle alors de subdivision à pas constant.

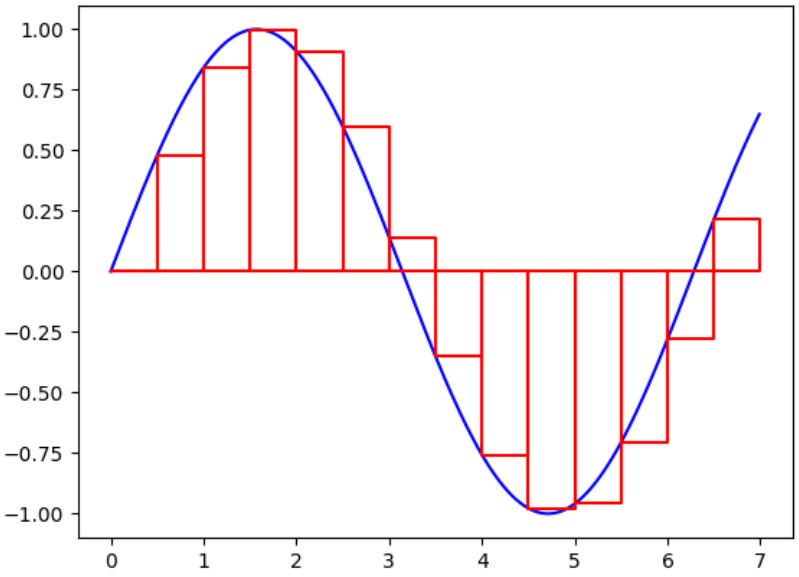


Figure 6.1 – Approximation de la fonction *sin* avec un pas de 0.5

6.5.2 Terminaison

Montrer la terminaison ne pose pas de problème : l'algorithme repose sur une boucle itérative finie et aucune opération illicite n'est effectuée.

6.5.3 Algorithme

```
def rectangle(f, a, b, n):
    # l'aire vaut zéro au début
    integrale = 0

    # le pas étant constant, la largeur est la même partout
    largeur = (b - a) / n

    for k in range(n):
        # on calcule la hauteur
        hauteur = f(a + k * largeur)

        # on ajoute l'aire du rectangle à l'intérgrale
        integrale += hauteur * largeur

    # on renvoie la somme des aires des rectangles
    return integrale
```

6.5.4 Méthode des trapèzes

La méthode est sensiblement la même, mais on approxime la fonction par des fonctions affines et non plus des constantes.

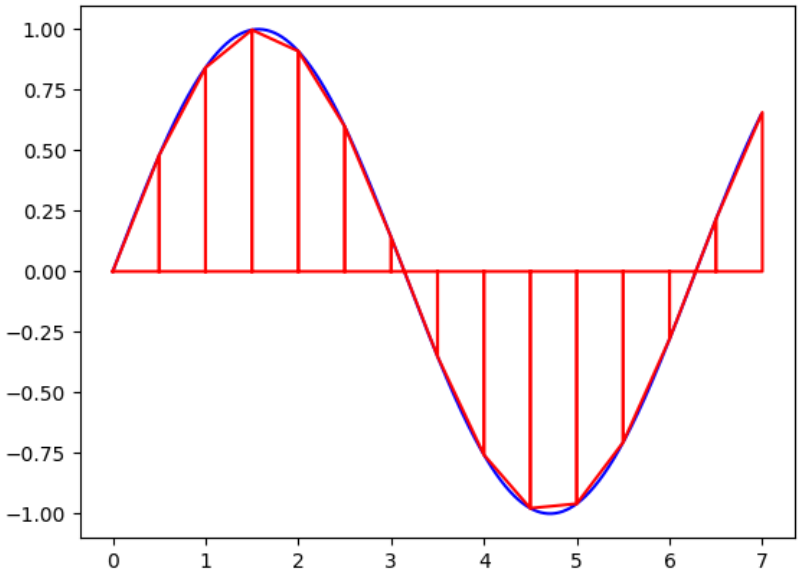


Figure 6.2 – Approximation de la fonction *sin* avec un pas de 0.5

La largeur ne change pas par rapport à la méthode des rectangles, mais la hauteur devient la moyenne entre $f(a + k * largeur)$ et $f(a + (k + 1) * largeur)$. Ainsi :

```
def trapeze(f, a, b, n):
    integrale = 0
    largeur = (b - a) / n

    for k in range(n):
        hauteur = f(a + k * largeur) + f(a + (k+1) * largeur)
        integrale += largeur * hauteur / 2

    return integrale
```

6.6 Méthode d'Euler : résolution d'une équation différentielle

6.6.1 Principe

Cette méthode permet de résoudre des équations différentielles d'ordre 1 de la forme :

$$\forall t \in [a\ ;\ b],\ y'(t) = f(t, y(t))$$

dans lesquelles on cherche y.

On considère les conditions initiales : $(t_0\ ;\ y_0)$ telles que $y_0 = y(t_0)$, et un intervalle $[0\ ;\ T]$ dans lequel va varier t. L'enjeu est de prendre des valeurs suffisamment proches tout en conservant un temps de calcul raisonnable, à l'instar de la méthode des rectangles, nous allons subdiviser l'intervalle $[a\ ;\ b]$ en N sous-intervalles de longeur : $h = \dfrac{b-a}{N}$ [3]. On définit alors les

3. h est le pas et l'on a : $t_{n+1} = t_n + h$

valeurs de t_n pour lesquelles on va approximer $y(t_n)$ par y_n :

$$\forall n \in [\![0\ ;\ N]\!],\ t_n = nh$$

Ainsi, nous allons avoir une suite $(y_n)_{n\in[\![0\ ;\ N]\!]}$ et :

$$\forall n \in [\![0\ ;\ N]\!],\ y_n \approx y(t_n)$$

Les y_n sont calculables de proche en proche grâce à la formule de récurrence :

$$\forall n \in [\![0\ ;\ N]\!],\ y_{n+1} = y_n + h \cdot f(t, y_n)$$

Il restera enfin à tracer la courbe des y_n (voir chapitre 4).

6.6.2 Terminaison

Comme pour la méthode des rectangles, il n'y a qu'une boucle for, ni aucune opération illicite, donc le programme termine.

6.6.3 Algorithme

Avec des listes Python :

```
1  def euler(f, a, b, N, y0):
2      # on calcule le pas
3      h = (b - a) / N
4
5      # construction des abscisses
6      X = [a + n * h for n in range(N)]
7
8      # on simplifie un peu le problème en considérant y(a) = y0
9      Y = [y0]
10
11     # N - 1 car la première valeur (condition initiale) est déjà
       ↪ dans la liste
12     for i in range(N - 1):
13         # on extrait le tn et yn
14         t, y = X[i], Y[i]
15
16         # calcul de y_{n+1}
17         y = y + h * f(t, y)
18
19         # on stocke la nouvelle valeur de y (y_{n+1})
20         Y.append(y)
21
22     # on renvoie les points calculés (pour tracer une courbe par
       ↪ exemple)
23     return X, Y
```

On peut tout à fait raccourcir la boucle `for` en :

```
for i in range(N - 1):
    Y.append(Y[i] + h * f(X[i], Y[i]))
```

Avec des `array` (voir 4.2) :

```
def euler(f, a, b, N, y0):
    X = np.linspace(a, b, N)
    Y = np.zeros(N)

    # condition initiale y(a) = y0
    Y[0] = y0

    # calcul du pas
    h = (b - a) / N

    # calcul des abcisses
    for i in range(N):
        Y[i + 1] = Y[i] + h * f(X[i], Y[i])

    return X, Y
```

6.6.4 Application

(Corrigé : 8.3.1)
On cherche à résoudre l'équation différentielle :

$$\forall t \in [-1\ ;\ 10],\ (t^2+1)y' + (t-1)^2 y = t^3 - t^2 + t + 1$$

Avec la condition initiale : $y(-1) = 8$. Le but est de reprogrammer l'algorithme, pas d'utiliser les fonctions.

Les grandes étapes :

1. Il faut commencer par isoler y' pour trouver f, une fonction de t et de y.
2. Construire le tableau des ordonnées.
3. Construire les abcisses de la solution pour une condition initiale donnée.
4. Tracer la courbe de la solution.
5. (Juste parce que c'est joli, recommencer avec une autre condition initiale.)

6.7 Exponentiation rapide

6.7.1 Principe

Dans toute cette sous-section, nous voulons élever $x \in \mathbb{R}$ à une puissance $p \in \mathbb{N}^*$. Partons de l'algorithme naïf, récursif d'exponentiation :

```
def puissance(x, p):
    if p == 0: return 1
    else: return x * puissance(x, p - 1)
```

Ainsi, cet algorithme va effectuer p tours de boucles avant de terminer. Sa complexité est donc un $\Theta(p)$. Cette complexité n'est pas catastrophique, mais on peut mieux faire. Pour cela il faut remarquer trois choses :

— si $p = 0$, $x^p = 1$

— si p est pair, $x^p = (x^2)^{\frac{p}{2}}$

— si p est impair, $x^p = x \times (x^2)^{\frac{p-1}{2}}$

Ainsi, si p est pair, on peut calculer $y^{\frac{p}{2}}$ avec $y = x^2$, si p est impair, on calcule $y^{\frac{p-1}{2}}$ avec $y = x^2$ et on multiplie par x.

La différence peut sembler futile, mais si on représente le nombre de multiplication effectuées pour calculer les puissances de 5 de 1 à 100 : On se rend compte que l'algorithme rapide est

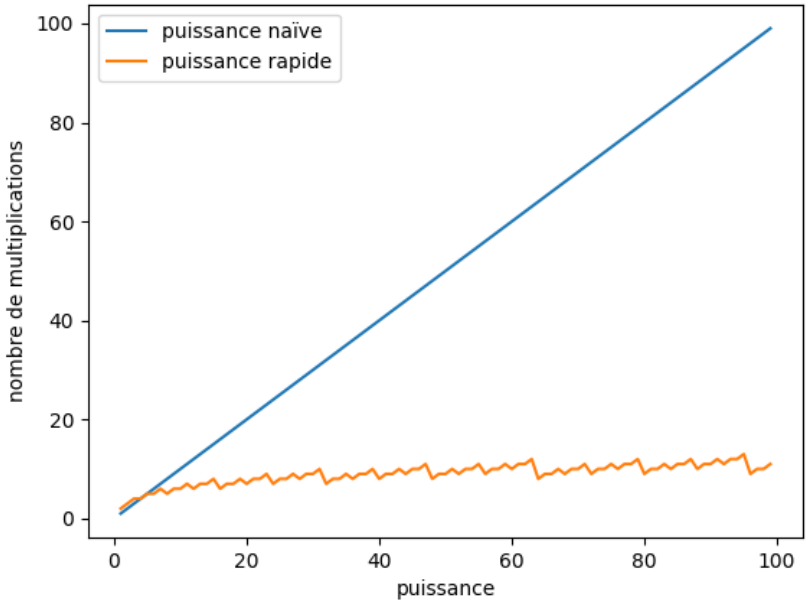


Figure 6.3 – Nombre de multiplications effectuées en fonctions de la puissance

réellement plus performant que l'algorithme naïf. De plus, on peut montrer par le calcul que l'exponentiation rapide a une complexité en $\Theta(\log n)$.

6.7.2 Terminaison

L'algorithme termine car p est un entier postif, qui décroît à chaque tour de boucle (il est divisé par deux à chaque fois). Donc d'après le principe de descente infinie de Fermat, p va finir par être nul.

6.7.3 Algorithme

On peut écrire une version récursive :

```
1 def puissance_rapide(x, p):
2     if p == 0: return 1
3     elif p % 2: return x * puissance(x**2, (p - 1) // 2) # si p %
      ↪  2 != 0 donc si p est impair
4     else: return puissance(x**2, p // 2)
```

6.8 Binaire et changement de bases

6.8.1 Principe

Dans les ordinateurs, les nombres sont représentés par une représentation dite en "binaire". Il s'agit d'une succession de 0 et de 1 qui correspondent au nombre. De manière courante nous utilisons la notation en base 10. L'enjeux de cette partie est de voir comment baser du binaire au décimal et d'implémenter cet algorithme. Dans la représentation binaire, on appelle nombre de bits le nombre de chiffres. Quelques exemples sur deux bits :

décimal	binaire
0	00
1	01
2	10
3	11

Pour passer du binaire au décimal, il faut écrire chaque chiffre du nombre ainsi que la puissance de 2 auquel il est associé. La méthode générale est de partir du chiffre le plus à gauche et de lui associé la puissance 2^0, le chiffre juste à droite reçoit reçoit la puissance 2^1 et ainsi de suite. Par exemple, si l'on prend le nombre écrit en binaire 1011 :

$$\begin{array}{cccc} 2^3 & 2^2 & 2^1 & 2^0 \\ 1 & 0 & 1 & 1 \end{array}$$

Ainsi 1011 en binaire, peut se décomposer en décimal comme $1\times 2^3+0\times 2^2+1\times 2^1+1\times 2^0$, d'où :

$$\begin{aligned} 1\times 2^3+0\times 2^2+1\times 2^1+1\times 2^0 &= 1\times 8+0\times 4+1\times 2+1\times 1 \\ &= 8+2+1 \\ &= 11 \end{aligned}$$

Ainsi le nombre 1011 en binaire est équivalent au nombre 11 en décimal.

Pour passer du décimal au binaire, il faut décomposer le nombre en décimal en effectuant des disivions euclidiennes par deux et en gardant les restes tant que le quotient est non nul. Reprenons 11 comme point de départ.

— on divise 11 par 2. $11 = 2\times 5+1$, on garde le 1

— $5 = 2\times 2+1$, on place ce deuxième reste (1) à droite du premier. (le nombre décomposé en binaire ressemble alors à : ..11).

— $2 = 2\times 1+0$ donc on met le 0 à droite des deux premiers chiffres et on continue car 1 est non nul

— $1 = 2\times 0+1$ on place alors ce dernier reste à droite des trois premiers chiffres et on s'arrête car le quotient est nul.

On retrouve alors le nombre 1011 en binaire.

6.8.2 Terminaison

Dans le cas du passage du binaire vers le décimal, l'algorithme va devoir parcourir tous les chiffres du nombre, la boucle `for` ne contenant pas d'opérations illicites, l'algorithme est fini.

Pour le cas de la conversion, l'algorithme boucle tant que le quotient est non nul, or le quotient est divisé par deux au tour suivant. Ainsi la suite des quotients au fil des tours de boucle forme une suite d'entiers positifs décroissantes, donc il existe un quotient nul, d'où la terminaison de l'algorithme.

6.8.3 Algorithme

Pour passer du binaire au décimale, la fonction va prendre en argument le nombre au format binaire dans une chaîne de caractère et appliquer le principe vu plus haut, de manière à renvoyer un entier en base 10.

```
def decimal(nb_binaire):
    resultat = 0
    for i in range(len(nb_binaire)):
        chiffre = int(nb_binaire[-(i + 1)] # on extrait le
        ↪ chiffre d'indice i en partant de la fin
        puissance_deux = 2 ** i # on calcule la puissance de 2
        ↪ correspondante
        resultat += chiffre * puissance_deux # on ajoute au
        ↪ résultat
    return resultat
```

Pour convertir un nombre du format décimal vers le binaire, nous allons faire une fonction qui va prendre en argument le nombre à convertir et qui va renvoyer la liste des chiffres en binaire.

```
def binaire(nb_decimal):
    resultat = [nb_decimal % 2] # on initialise la liste avec le
    ↪ premier reste
    nb_decimal //= 2 # on divise le nombre pour garder le
    ↪ quotient
    while nb_decimal: # tant que le quotient est non nul
        resultat.insert(0, nb_decimal % 2) # on insert à droite
        ↪ du premier élément, le reste suivant
        nb_decimal //= 2 # on calcule le quotient suivant
    return resultat # on renvoie la liste des restes
```

6.9 Tri par insertion

6.9.1 Principe

Le principe est de parcourir la liste, et tant que l'élément que l'on regarde est plus petit que l'élément précédent et que les indices sont positifs, on inverse les éléments.

En prenant n, la longueur de la liste, la complexité est en $\Theta(n)$ si la liste est déjà triée (complexité dans le meilleur cas), et peut atteindre $\Theta(n^2)$ dans le pire cas.

6.9.2 Terminaison

La boucle `while` termine car soit l'élément que l'on déplace est décalé jusqu'à la première case de la liste, soit l'élément se retrouve à sa place. La boucle `for` termine car on ne parcourt la liste qu'une seule fois.

6.9.3 Algorithme

```
def tri_insertion(liste):
    for i in range(1, len(liste)):
        element_a_trier = liste[i]
        j = i
        while j > 0 and element_a_trier < liste[j - 1]: # tant
         ↪ que 'element_a_trier' n'est pas à sa place et que
         ↪ l'indice est positif
            liste[j] = liste[j - 1] # on recule l'élément à trier
             ↪ dans la liste
            j -= 1 # on recule l'indice de l'élément comparé
        liste[j] = element_a_trier # on insert l'élément à trier
         ↪ à sa place
```

6.10 Tri-fusion

6.10.1 Principe

Le tri fusion repose sur un principe simple :

— une liste a un seul élément est triée

— pour trier une liste à n éléments on fusionne les deux sous-listes de longeurs égales extraites de cette liste

On va donc commencer par diviser la liste donnée en argument en plusieurs sous-listes jusqu'à obtenir des listes de longueur 1. On triera ensuite les listes en les fusionnant.

Le fonctionnement de la fusion est central. On note a et b les deux sous-listes à fusionner et a_i et b_i les $i^{\text{ième}}$ éléments des listes a et b. On prend un indice i associé à la liste a et un indice j associé à b.

On initialise i et j à 0, et on créée une liste vide c munie d'un indice k initialisé à 0 aussi.

Tant que i est inférieur à la taille de a et j est inférieur à la taille de b, on boucle suivant le principe suivant :

— si $a_i \leqslant b_j$, on ajoute a_i à c, on incrémente i de un

— si $a_i > b_k$, on ajoute b_j à la liste c, et on incrémente j de un

La liste c est alors la liste triée résultant de la fusion entre a et b.

La complexité du tri-fusion dans le pire cas se divise en trois parties : le tri de la première sous-liste, de la seconde sous-liste, et la fusion des deux. On peut alors montrer que si la liste donnée est de taille n la complexité est en $\Theta(n \log n)$.

6.10.2 Terminaison

L'algorithme de fusion est terminé, puisqu'à chaque tour, soit i soit j augmente d'un, donc on finit par atteindre la taille des sous-listes. e La fonction principale va découper la liste en sous-liste de longeur $\frac{n}{2}$ à chaque tour, donc on finira par avoir des listes de longueurs un qui sont, par définition, déjà triées.

6.10.3 Algorithme

```
def fusion(a, b):
    i = j = 0
    c = []
    while i < len(a) or j < len(j):
        if j == len(b) or (i < len(a) and a[i] <= b[j]):
            c.append(a[i])
            i += 1
        elif i == len(a) or (j < len(b) and a[i] > b[j]):
            c.append(b[j])
            j += 1
    return c

def tri_fusion(liste):
    n = len(liste)
    if n == 1: return liste
    else: return fusion(tri(liste[n // 2: ]), tri(liste[: n //
     ↪  2]))
```

Chapitre 7

Programmation orientée objet

7.1 Introduction à la programmation orientée objet (POO)

7.1.1 Définition d'un objet et principe de la POO

En Python, toutes les variables sont typées (i.e. ont un type). Ainsi lorsque l'on écrit : `variable = 4`, on crée en fait un objet `variable`, de type `int` et qui contient `4` (la valeur elle-même a aussi le statut d'objet).

On dit que `variable` est une instance du type `int`. Ce qui explique pourquoi `4` a également le statut d'objet : il s'agit bien d'une instance aussi.

De manière un peu plus formelle, une instance est un représentant d'un type donné. Le type est une notion abstraite (un entier, une liste etc) alors que l'instance est un objet particulier rattaché à cette notion (`4` est un entier en particulier, et non un entier en tant que notion).

La programmation orientée objet va avoir pour objectif de créer de nouveaux types de variables. On pourra ensuite écrire `ma_variable = MonType()` où `ma_variable` sera une instance (un objet) du type `MonType`.

7.1.2 Classes, méthodes et attributs

Première approche pratique

Reprenons un exemple : `ma_liste = [1, 2, 3]`. Ici, `ma_liste` est une instance du type `list`. On dit qu'on accède à une méthode ou à un attribut d'une instance lorsqu'on utilise la syntaxe avec le point : `mon_instance.methode()`, ou `mon_instance.attribut`.

Sur les listes par exemple, lorsque l'on écrit : `ma_liste.append(2)`, on appelle en fait la méthode `append` associée au type `list` (toutes les listes ont cette méthode), avec l'argument `3`. Les méthodes sont de ce fait des fonctions associées à un type précis.

Les attributs correspondent à des variables propre à un type. Il y a assez peu d'exemple en Python, on pourra toutefois citer les tableaux `numpy` qui ont un attribut `shape` qui correspond à la taille du tableau.

7.1.3 Formalisation

En résumé :

Une méthode est une fonction associée à un type précis.

Un attribut est une variable associée à un type précis.

Et on appelle une méthode (ou on demande la valeur d'un attribut) grâce au point que l'on place entre l'instance et la méthode (ou l'attribut).

Donc pour l'instant, on a vu qu'un type est en fait constitué de fonctions (méthodes) et de variables (attributs). Pour programmer un nouveau type, il faut indiquer explicitement à Python quelle fonction (ou variable) est une méthode (ou attribut) du nouveau type. Pour cela on utilise une classe. La syntaxe est assez simple :

```
class MonType:
    attribut = <valeur>
    def methode(<argument>):
        <actions>
        return <variable>
```

7.2 Classes et types

7.2.1 Créer un nouveau type

Créer un nouveau type de variable est équivalent à créer une classe. Il suffit dont d'écrire :

```
class MonType:
    pass
```

Le mot-clef `pass` permet de dire à Python qu'il n'y a rien à faire. [1] Dans la suite nous verrons comment remplir cette classe avec des méthodes et des attibuts.

On peut ensuite écrire : `ma_variable = MonType()` pour créer une instance du type `MonType`.

7.2.2 Implémentation de méthodes et d'attributs

Pour définir une nouvelle méthode, il suffit d'écrire une fonction dans la classe. Cette fonction peut prendre des arguments et renvoyer (ou afficher) un résultat, comme n'importe quelle fonction.

La seule subtilité est que toutes les méthodes doivent prendre au moins un argument : l'instance elle-même. Explications : lorsque l'on écrit : `ma_variable = MonType()`, on crée une instance qui porte le nom `ma_variable`. Les méthodes vont donc prendre cette instance en premier argument, qui est nommé `self` par convention.

Si l'on reprend notre exemple :

```
class MonType:
    def methode(self):
        return 42
```

On peut ensuite créer une instance pour voir le résultat (limité pour l'instant) :

```
>>> variable = MonType()
>>> variable.methode()
42
```

1. i.e. que la classe est vide, que la fonction n'effectue aucun calcul etc.

Pour créer un attribut, ce n'est pas plus compliqué qu'une simple affectation.

```
class MonType:
    def methode(self):
        self.attribut = 0
        return 42
```

Bien que ce code soit fonctionnel, nous rencontrons quelques paradoxes : l'attribut est défini à l'intérieur d'une méthode, si bien que tant que cette méthode n'a pas été exécutée, l'attribut n'existe pas. Ainsi :

```
>>> variable = MonType()
>>> variable.attribut
AttributeError
>>> variable.methode()
42
>>> variable.attribut
0
```

On voudrait pourtant pouvoir utiliser l'attribut comme une propriété de l'instance indépendamment des méthodes exécutées. La méthode spéciale `__init__` permet de pallier ces problèmes de définition.

7.2.3 Méthodes spéciales

Python permet de nombreuses manipulations avec les objets comme la re-définition des opérateurs pour le nouveau type (+, -, *, /, ...), mais aussi la possibilité de rendre l'objet "appelable", ou "itérable". Ces notions seronts vues en même temps que les méthodes spéciales correspondantes.

Avant de voir ces méthodes spéciales, un petit point de syntaxe, dans une classe, les méthodes spéciales sont toujours de la forme `__methode__(self, <argument>):`.

Initialiser un objet

La méthode spéciale `__init__` se lance automatiquement à la création de l'instance. Ainsi, lorsque l'on écrit : `variable = MonType()`, la méthode `__init__` se lance.

Il faut voir la création de l'instance comme un appel à `__init__`, le premier argument est l'instance elle-même (`self` par convention), les arguments suivants sont des arguments normaux. Par exemple, si l'on définit la classe par :

```
class MonType:
    def __init__(self, attribut):
        self.attribut = attribut
```

On peut ensuite s'en servir de la manière suivante :

```
>>> variable = MonType(1)
>>> variable.attribut
1
```

Contrairement à l'attribut défini dans l'exemple du paragraphe précédent, cet attribut pourra être utilisé dans toutes les méthodes de la classe sans problème.

Redéfinition des opérateurs

Nous ne verrons ici que comment redéfinir les opérateurs élémentaires, à savoir l'addition, la soustraction, la multiplication et la division.

Un peu de théorie avant la suite, lorsque l'on écrit un calcul, Python lance en fait une méthode qui correspond au calcul à effectuer. Ainsi `2+3` exécute une méthode sur les `int`, `"bon" + "jour"` lance la même méthode, mais sur les `str`.

Un peu plus de subtilité, pour chaque opérateur, il existe deux méthodes qui correspondent aux deux syntaxes :

— `variable <operateur> <argument>` qui est la syntaxe normale, avec l'instance à gauche.

— `<argument> <operateur> variable` qui est la syntaxe inverse avec l'instance à droite.

La liste des principaux opérateurs :

— L'addition correspond à la méthode spéciale : `__add__(self, argument):`. Cette méthode est appelée avec la syntaxe : `variable + <argument>`. Dans les faits, la commande : `a = b + c` devient : `a = b.__add__(c)` [2].

— La soustraction peut-être implémentée avec la méthode `__sub__(self, argument):`.

— La multipliciation par `__mul__(self, argument):`.

— Et la division par `__truediv__(self, argument):`.

Les méthodes sont les syntaxes inverses ont les même noms précédés d'un "r". Ainsi, la syntaxe inverse pour l'addition correspond à la méthode spéciale : `__radd__(self, argument):`.

En général, les opérateurs sont symétriques, on peut donc définir les méthodes des syntaxes inverses par la syntaxe : `__roperateur__ = __operateur__`.

Par exemple :

```
class MonType:
    def __init__(self, attribut):
        self.attribut = attribut

    def __add_(self, nombre):
        return self.attribut + nombre

    def __radd_(self, nombre):
        return self.attribut * nombre
```

Et en pratique :

```
>>> variable = MonType(2)
>>> variable + 5 # __add__
7
>>> 2 + variable # __radd__
10
```

2. Le principe est le même pour les autres opérateurs

Appeler une instance de la classe

Lorsque l'on appelle une fonction, on utilise une syntaxe avec des parenthèses. L'idée est la même ici, on va rendre les instances de notre classe "appelables" ainsi, lorsque l'on écrira : `variable(<arguments>)`, Python exécutera une méthode avec les arguments donnés.

La méthode en question est : `__call__(self, <arguments>):`. Un petit exemple pour comprendre :

```
class MonType:
    def __init__(self, attribut):
        self.attribut = attribut

    def __call__(self, nombre):
        return self.attribut * nombre
```

Et l'exécution :

```
>>> variable = MonType(2)
>>> variable(5) # 2 * 5
10
```

Rendre une classe indexable

Un exemple connu d'objet indexable sont les listes, ou les chaînes de caractères. Plus généralement les instances de la classe vont réagir à la syntaxe : `variable[<arguments>]`. On peut mettre à peu près tout et n'importe quoi dans les crochets.

La méthode spéciale est ici : `__getitem__(self, arguments):`. Comme tout à l'heure, un exemple :

```
class MonType:
    def __init__(self, attribut):
        self.attribut = attribut

    def __getitem__(self, nb):
        return self.attribut * nb
```

Et avec un exemple d'utilisation :

```
>>> variable = MonType(5)
>>> variable[2]
10
```

Afficher un objet

Il y a deux manière d'afficher un objet :

— En programmant le transtypage vers le type `str` via la méthode spéciale : `__str__(self)`, on affiche alors via la fonction `print`.

— En programmant la représentation de l'objet via : `__repr__(self)`, l'affichage se fait alors par un simple appel à la variable.

Dans les deux cas, la fonction va devoir renvoyer la chaîne de caractères.
Premier exemple avec le transtypage :

```
class MonType:
    def __init__(self, attribut):
        self.attribut = attribut

    def __str__(self):
        return str(self.attribut) # on renvoie la représentation
         ↪  sous forme de chaîne de caractères
```

En pratique :

```
>>> variable = MonType(2)
>>> print(variable)
2
```

Second exemple avec la représentation :

```
class MonType:
    def __init__(self, attribut):
        self.attribut = attribut

    def __repr__(self):
        return str(self.attribut)
```

Et :

```
>>> variable = MonType(2)
>>> variable
2
```

Récapitulatif

Méthode spéciale	Effet
`__init__`	Initialiser l'objet
`__add__`	Addition
`__sub__`	Soustraction
`__mul__`	Multiplication
`__truediv__`	Division
`__call__`	Rendre une classe appelable
`__getitem__`	Rendre une classe indexable
`__str__` et `__repr__`	Afficher un objet

7.2.4 Héritages

En Python, et dans les autres langages orientés objets aussi, une classe peut hériter des méthodes et attributs d'une autre classe. Ainsi, la nouvelle classe contiendra toute les méthodes

de la classe dont elle a héritée plus de nouvelles méthode (ou attributs) qui seront rajoutés "par dessus" la classe d'origine.

Pour faire en sorte qu'une classe `ClassA` hérite d'une classe `ClassB`, on écrit : `class ClassA(ClassB)`.

Ici, nous allons nous intéresser à un exemple un peu plus concret que ceux vu plus haut dans ce chapitre : nous allons créer un nouveau type de chaînes de caractères, basé sur les `str`. Nous allons donc commencer par :

```
class Str(str): # Python est sensible à la casse, donc pas de risque
    def __init__(self, valeur):
        self.data = valeur

    def methode(self):
        return 42
```

Ici, les instances de classe `Str` vont se comporter exactement comme des chaînes de caractères normale, mais on peut leur appliquer la méthode `methode()` qui renvoie 42.

```
>>> variable = Str("Ceci est une chaîne")
>>> variable.methode()
42
```

On peut également s'amuser à redéfinir les opérateurs sur les entiers :

```
class Int(int):
    def __init__(self, valeur):
        self.valeur = valeur

    def __add__(self, b):
        return self.valeur * b

    def __sub__(self, b):
        return self.valeur + b

    def __mul__(self, b):
        return self.valeur / b

    def __truediv__(self, b):
        return self.valeur - b
```

Et pour montrer les effets :

```
>>> var_a = Int(2)
>>> var_b = Int(5)
>>> var_a + var_b
10
>>> var_a # __repr__ n'est pas programmée, mais l'héritage fait que la méthode fonctionne
```

```
2
>>> var_a / var_b
7
```

7.3 Mise en pratique

Il y a beaucoup d'objets mathématiques qui ne sont pas programmé en Python, ce qui nous fournit l'occasion de quelques exemples.

7.3.1 Vecteurs

(Corrigé : 8.4.1) Un premier exemple de programmation orientée objet : les vecteurs. Le but va être de construire un objet `Vecteur` qui permettra les opérations suivantes :

— addition de deux vecteurs

— multiplication par un scalaire

On pourra afficher le vecteur comme un tuple.

7.3.2 Polynômes

(Corrigé : 8.4.2) Cet exemple, par rapport au précédent, est plus complexe d'un point de vue mathématiques car les opérations sur les polynômes sont plus délicates à programmer que les opérations sur les vecteurs. Mais d'un point de vue informatique, les fonctions et méthodes entrant en jeux sont du même ordre de grandeur en termes de difficulté.

La classe `Polynome` devra :

— gérer les opérations élémentaires sur les polynômes (addition et soustraction)

— permettre un affichage du polynôme (sous la forme `c + bX + aX^2`)

— gérer l'opération de dérivation via une méthode à part.

— permettre l'évaluation du polynôme en une valeur précise.

Quelques rappels pour donner des pistes ou aider à mieux comprendre l'objet mathématique :

1. Un polynôme s'écrit sous la forme : $\sum_{i=0}^{n} a_i \cdot X^i$ où n est son degré et où $(a_n)_{n\in[\![0\ ;\ n]\!]}$ sont ses coefficients
2. Un polynôme est entièrement défini par la donnée de ses coefficients.
3. L'addition et la soustraction se font terme à terme et le degré du résultat correspond au degré maximal des deux polynômes.
4. La dérivation a une formule explicite : $\sum_{i=1}^{n} i a_i \cdot X^{i-1}$.

Chapitre 8

Corrigés des applications

8.1 Fonctions et modules

8.1.1 Théorème de Pythagore

(Énoncé : 2.3.1)
On a le programme suivant :

```
# On créé une fonction qui prend trois arguments
def pythagore(a, b, c):
    # Si la somme des carrés des deux côtés est égale au carré du
    ↪ troisième
    if (a ** 2) + (b ** 2) == (c ** 2):
        return True
    # Sinon
    else:
        return False
```

8.1.2 Implémentation de la fonction factorielle

(Énoncé : 2.3.2)
Il y a n tour de boucle à faire, l'utilisation d'une boucle itérative s'impose :

```
def factorielle(n):
    resultat = 1
    for i in range(1, n + 1):
        resultat = resultat * i
    return resultat
```

8.1.3 Test de la primalité d'un nombre

(Énoncé : 2.3.3)
En reprenant les étapes données, on obtient :

```
from math import sqrt

def premier(p):
```

```
    for n in range(2, int(sqrt(p)) + 1):
        if p % n == 0:
            return False
    return True
```

8.1.4 Version récursive de l'algorithme d'Euclide

(Énoncé : 2.3.4)

Le programme récursif va s'axer autour de la condition $b = 0$. En effet si b est nul, le PGCD de a et b est a, sinon $PGCD(a\ ;\ b) = PGCD(b\ ;\ a\ MOD\ b)$. Ainsi :

```
def pgcd(a, b):
    if b == 0:
        return a
    else:
        return pgcd(a, a % b)
```

8.2 Outils pour l'ingénierie numérique

8.2.1 Calcul des points d'une fonction et affichage

(Énoncé : 4.3.1)

En choisissant de représenter $f\ :\ x \mapsto \dfrac{\sin(x)}{x}$ sur l'intervalle $[0.5\ ;\ 5]$ avec 1000 points :

```
from math import sin
import matplotlib.pyplot as plt
import numpy as np

# Définition de la fonction f
def f(x):
    return sin(x) / x

# Calcul automatique des abcisses
X = np.linspace(0.5, 5, 1000)

# Construction des ordonnées
Y = np.vectorize(f)(X)

# Affichage du résultat
plt.plot(X, Y)
plt.show()
```

8.3 Algorithmes usuels

8.3.1 Courbe d'une solution d'une équation différentielle

(Énoncé : 6.6.4)

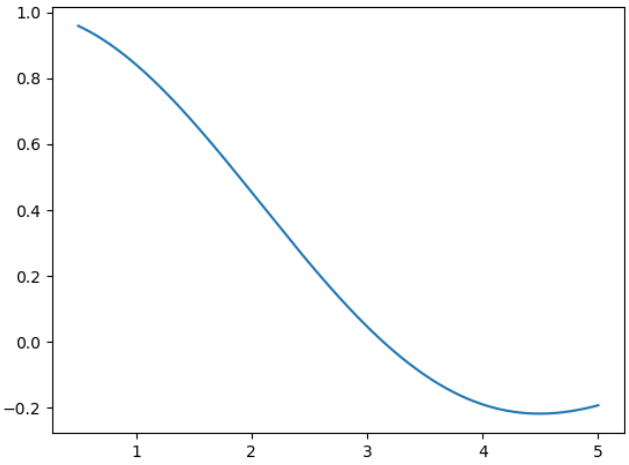


Figure 8.1 – $f(x)$ sur $[0.5\ ;\ 5]$

On veut tracer une solution de cette équation différentielle :

$$\forall t \in [-1\ ;\ 10],\ (t^2+1)y' + (t-1)^2 y = t^3 - t^2 + t + 1$$

Avec la condition initiale : $y(-1) = 8$.

En reprenant point par point :

1. Assez simplement, on isole y' :

$$\begin{aligned}(t^2+1)y' + (t-1)^2 y &= t^3 - t^2 + t + 1\\ (t^2+1)y' &= t^3 - t^2 + t + 1 - (t-1)^2 y\\ y' &= \frac{1}{(t^2+1)} \cdot (t^3 - t^2 + t + 1 - (t-1)^2 y)\end{aligned}$$

   Ainsi, on pose $\forall t \in [-1\ ;\ 10],\ y' = f(t,y) = \dfrac{1}{(t^2+1)} \cdot (t^3 - t^2 + t + 1 - (t-1)^2 y)$

```
def f(t, y):
    return (1 / (t**2 + 1)) * (t**3 - t**2 + t + 1 - (t - 1)**2 * y)
```

2. On se place sur l'intervalle $[-1\ ;\ 10]$ avec un nombre arbitraire de points, on peut prendre $N = 1000$. Ainsi,

```
h = (10 + 1) / 1000
X = [-1 + n * h for n in range(1000)]
```

3. On initialise les ordonnées avec la condition initiale et on complète la liste :

```
Y = [8]
for i in range(1000 - 1):
    Y.append(Y[i] + h * f(X[i], Y[i]))
```

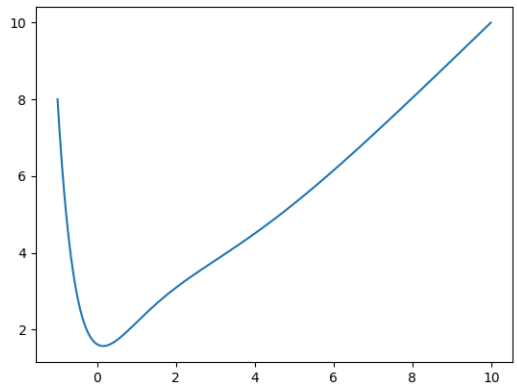
4. Il ne reste plus qu'à tracer et à afficher :

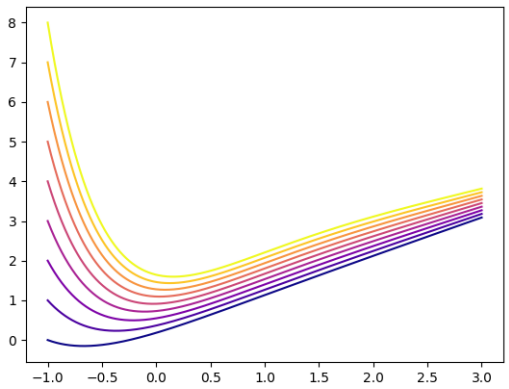
```
plt.plot(X, Y)
plt.show()
```

En résumé, le code complet :

```
import matplotlib.pyplot as plt

def f(t, y):
    return (1 / (t**2 + 1)) * (t**3 - t**2 + t + 1 - (t - 1)**2 *
    ↪  y)

h = (10 + 1) / 1000
X = [-1 + n * h for n in range(1000)]

Y = [8]
for i in range(1000 - 1):
    Y.append(Y[i] + h * f(X[i], Y[i]))

plt.plot(X, Y)
plt.show()
```

La figure de gauche représente la solution de condition initiale $y(-1) = 8$ sur l'intervalle $[-1 ; 10]$ et la figure de droite représente les solutions approchées pour des valeurs initiales entre 0 et 8 et t entre -1 et 3.





8.4 Programmation orientée objet

8.4.1 Vecteurs

(Énoncé : 7.3.1)

Initialiser l'objet

Le plus simple pour représenter un vecteur est d'utiliser une liste Python contenant les composantes du vecteurs selon les différentes coordonnées. Ainsi l'initialisation du vecteur va ressembler à :

```
class Vecteur:
    def __init__(self, composantes):
        self.composantes = composantes
        self.taille = len(composantes) # on stocke aussi la
         ↪ dimension du vecteur
```

Afficher l'objet

Commencer par programmer l'affichage du nouvel objet permet de faire des tests assez rapidement ce qui est agréable. Ici rien de très compliqué.

On pourrait transtyper deux fois notre liste de coordonnées : une première fois pour avoir un tuple et une seconde fois pour avoir une chaîne de caractères. Cette solution fonctionne, mais n'est pas très élégante. Il est préférable de construire directement la représentation de l'objet :

```
def __repr__(self):
    composantes_str = [str(i) for i in self.composantes] # on
     ↪ construit la liste des composantes sous forme de chaînes
     ↪ de caractères
    return "(" + ", ".join(composantes_str) + ")"
```

Addition de deux vecteurs

L'addition sur les vecteurs est une addition terme à terme, donc rien de très compliqué :

```
def __add__(self, vecteur):
    if self.taille != vecteur.taille:
        raise ValueError("on ne peut pas sommer deux vecteurs de
         ↪ tailles différentes") # On arrête l'exécution du
         ↪ programme et on renvoie une erreur de valeur

    resultat = [] # les composantes du nouveau vecteur
    for i in range(self.taille):
        resultat.append(self.composantes[i] +
         ↪ vecteur.composantes[i])

    return Vecteur(resultat) # on renvoie une instance de la
     ↪ classe Vecteur
```

Multiplication par un scalaire

La multiplication entre un vecteur et un scalaire est simple : chaque composante du vecteur est multipliée par le scalaire.

```
def __mul__(self, scalaire):
    if not isinstance(scalaire, (int, float)):
        raise TypeError("ce n'est pas un scalaire") # si
         ↪ 'sclaire' n'est ni un int, ni un float, on arrête
         ↪ l'exécution du programme et on renvoie une erreur de
         ↪ type
```

```
4
5       resultat = []
6       for i in range(self.taille):
7           resultat.append(scalaire * self.composantes[i])
8
9       return Vecteur(resultat)
```

Résumé et idées pour aller plus loin

Le code complet du vecteur est donc :

```
1   class Vecteur:
2       def __init__(self, composantes):
3           self.composantes = composantes
4           self.taille = len(composantes)
5
6       def __repr__(self):
7           composantes_str = [str(i) for i in self.composantes]
8           return "(" + ", ".join(composantes_str) + ")"
9
10      __str__ = __repr__
11
12      def __add__(self, vecteur):
13          if self.taille != vecteur.taille:
14              raise ValueError("on ne peut pas sommer deux vecteurs
                ↪  de tailles différentes")
15
16          resultat = []
17          for i in range(self.taille):
18              resultat.append(self.composantes[i] +
                ↪  vecteur.composantes[i])
19
20          return Vecteur(resultat)
21
22      __radd__ = __add__
23
24      def __mul__(self, scalaire):
25          if not isinstance(scalaire, (int, float)):
26              raise TypeError("ce n'est pas un scalaire")
27
28          resultat = []
29          for i in range(self.taille):
30              resultat.append(scalaire * self.composantes[i])
31
32          return Vecteur(resultat)
33
34      __rmul__ = __mul__
```

Bien sûr il existe d'autres manipulations sur les vecteurs : produit scalaire, norme, produit vectoriel etc. Ces fonctions ne sont pas expliquées ici, mais sont tout à fait faisables.

8.4.2 Polynômes

(Énoncé : 7.3.2)

Initialiser l'objet

Avant d'écrire les méthodes, il faut penser à la manière dont l'objet va être représenté dans la classe. Ici, un polynôme est entièrement caractérisé par la donnée de ses coefficients, ainsi le polynôme va devoir avoir un attribut `coefficients` qui sera une liste. Le premier élément de la liste correspondra au coefficient de puissance nulle (indice 0 dans la liste), le second élément (indice 1) sera le coefficient de degré 1 etc.

On impose par ailleurs un attribut `deg` qui correspond au degré du polynôme. Le degré correspond à la puissance maximale, de coefficient non nul. En supposant que la liste ne finit pas par un 0, il s'agit de la longueur de la liste moins un.

Avec les deux derniers paragraphes, on peut déjà rédiger la méthode `__init__` :

```
class Polynome:
    def __init__(self, coefficients):
        self.coefficients = coefficients
        self.deg = len(coefficients) - 1
```

Afficher un polynôme

L'affichage maintenant. La liste des coefficients est triée par ordre croissant de puissance, donc le polynôme va être affiché dans cet ordre (on aurait pu l'afficher dans l'autre sens, mais cela ne présente pas beaucoup d'intérêt ici). L'idée va être de décomposer le polynôme en une liste de chaîne de caractères où chaque élément est un monôme qui compose le polynôme.

Nous allons donc reccourir à une liste `monomes`. Ensuite, il va falloir faire une boucle `for` qui balaye les coeffiicents du polynôme à afficher en distinguant les cas :

— si le coefficient vaut 0, il n'y a rien à afficher

— si la puissance vaut 0, $X^0 = 1$

— si la puissance vaut 1, il suffit d'afficher `X` et non `X^1`

— sinon, on affiche sous la forme `coefficient X^puissance`

On ajoute ensuite le monôme à la liste. Et enfin on utilise `join` pour intercaler un signe + entre chaque monôme. Ainsi :

```
    def __repr__(self):
    monomes = []
    for puissance in range(self.nb_coeff):
        if self.coefficients[puissance] != 0:
            if puissance == 0:
                monomes.append(str(self.coefficients[puissance]))
            elif puissance == 1:
                monomes.append(str(self.coefficients[puissance])
                 ↪  + "X")
            else:
                monomes.append(str(self.coefficients[puissance])
                 ↪  + "X^" + str(puissance))
    return " + ".join(monomes)
```

On peut également écrire : `__str__ = __repr__` pour pouvoir transtyper le polynôme vers une chaîne de caractères.

Addition et soustraction de polynômes

L'addition et la soustraction se ressemblent beaucoup. Le degré du résultat est égal au maximum des degrés. Dans les deux cas, l'idée est de construire une liste `nouveaux_coeff` qui va contenir les coefficients du résultat et à la fin, on renvoie le polynôme associé à ces coefficients. Il y a trois cas à distinguer :

— Si P n'a pas de coefficient de degré i, la somme est donc égale au coefficient de degré i de Q.

— De manière symétrique, si Q n'a pas de coefficient de degré i, la somme est alors égale au coefficient de degré i de P.

— Sinon, il s'agit de la somme des coefficients.

On obtient alors la code :

```
def __add__(self, P):
    nouveaux_coeff = []
    for i in range(max(self.deg, P.deg) + 1):
        if self.deg < i:
            nouveaux_coeff.append(P.coefficients[i])
        elif P.deg < i:
            nouveaux_coeff.append(self.coefficients[i])
        else:
            nouveaux_coeff.append(self.coefficients[i] +
             ↪  P.coefficients[i])
    return Polynome(nouveaux_coeff)
```

On procède de même avec la soustraction.

Évaluation du polynôme en une valeur

Il s'agit d'une méthode à part, assez simple. Il suffit de "remplacer" le X du polynôme par la valeur donnée. Nous allons commencer par définir une variable `resultat`, initialisée à 0 au début. On va ensuite balayer la liste des coefficients du polynômes. Par construction, l'indice du coefficient correspond à la puissance du monôme, ainsi :

```
def __call__(self, X):
    resultat = 0
    for puissance in range(self.deg + 1):
        resultat += self.coefficients[puissance] * X ** puissance

    return resultat
```

Dérivation d'un polynôme

En appliquant la formule explicite pour la dérivation :

```python
def derivee(self):
    nouveaux_coeff = [0]
    for puissance in range(self.deg + 1):
        if puissance:
            nouveaux_coeff[puissance - 1] =
             ↪  self.coefficients[puissance] * puissance
            nouveaux_coeff.append(0)
    return Polynome(nouveaux_coeff)
```

Code complet

```python
class Polynome:
    def __init__(self, coefficients):
        self.coefficients = coefficients

        self.deg = len(coefficients) - 1

    def __repr__(self):
        monomes = []
        for puissance in range(self.deg + 1):
            if self.coefficients[puissance] != 0:
                if puissance == 0:

                    ↪  monomes.append(str(self.coefficients[puissance]))
                elif puissance == 1:

                    ↪  monomes.append(str(self.coefficients[puissance])
                    ↪  + "X")
                else:

                    ↪  monomes.append(str(self.coefficients[puissance])
                    ↪  + "X^" + str(puissance))
        return " + ".join(monomes)

    __str__ = __repr__

    def __add__(self, P):
        nouveaux_coeff = []
        for i in range(max(self.deg, P.deg) + 1):
            if self.deg < i:
                nouveaux_coeff.append(P.coefficients[i])
            elif P.deg < i:
                nouveaux_coeff.append(self.coefficients[i])
            else:
                nouveaux_coeff.append(self.coefficients[i] +
                 ↪  P.coefficients[i])
        return Polynome(nouveaux_coeff)

    def __sub__(self, P):
```

```
33          nouveaux_coeff = []
34          for i in range(max(self.deg, P.deg) + 1):
35              if self.deg < i:
36                  nouveaux_coeff.append(-P.coefficients[i])
37              elif P.deg < i:
38                  nouveaux_coeff.append(self.coefficients[i])
39              else:
40                  nouveaux_coeff.append(self.coefficients[i] -
                    ↪  P.coefficients[i])
41          return Polynome(nouveaux_coeff)
42
43      def __call__(self, X):
44          resultat = 0
45          for puissance in range(self.deg + 1):
46              resultat += self.coefficients[puissance] * X **
                ↪  puissance
47
48          return resultat
49
50      def derivee(self):
51          nouveaux_coeff = [0]
52          for puissance in range(self.deg + 1):
53              if puissance:
54                  nouveaux_coeff[puissance - 1] =
                    ↪  self.coefficients[puissance] * puissance
55                  nouveaux_coeff.append(0)
56          return Polynome(nouveaux_coeff)
```