**Année scolaire 2025 – 2026**

Lycée Victor Hugo - Colomiers

# Terminale NSI

---

# LP
# Langages de programmation

---

**Thomas Ricart**
ricart.lvh@laposte.net

24 mars 2025

# LP01 – Mise au point d'un programme

## 1. Convention syntaxique

**Propriété 1: Les espaces**

- Il faut mettre un espace avant et après chaque opérateur.
- Pour les opérateurs mathématiques, on essaie de reconstituer les groupes de priorité (lorsqu'il y en a).

Code Python
```
# PAS BIEN
a=3
# BIEN
a = 3
```

Code Python
```
# PAS BIEN
x = 3 * 2 + 5
# BIEN
x = 3*2 + 5
```

- On ne met pas d'espace à intérieur des parenthèses, des crochets ou des accolades.
- Pour les virgules, et les deux points: pas d'espace avant mais une espace après.

Code Python
```
# PAS BIEN
for x in range( 5 ):
    print( 'bonjour' )
# BIEN
for x in range(5):
    print('bonjour')
```

Code Python
```
# PAS BIEN
if color == (0,255,0) :
    print('vert')
# BIEN
if color == (0, 255, 0):
    print('vert')
```

**Propriété 2: Les convention de nommage**

- Les variables à une lettre (comme `i, j, k`) sont réservées aux indices (notamment dans les boucles).
- Les autres variables doivent avoir des noms explicites, éventuellement écrits en `snake_case`.

Cas particulier des classes en Programmation Orientée Objet : leur nom doit commencer par une majuscule.

Code Python
```
# PAS BIEN
if d == 1:
    cep += vm
# BIEN
if date == 1:
    epargne += versement_mensuel
```

Code Python
```
# PAS BIEN
class voiture:
    #pass
# BIEN
class Voiture:
    #pass
```

## 2. Commentaires et docstrings

**Définition 1: Les docstrings**

Les `docstrings` sont des commentaires normalisés pour les fonctions, qui peuvent être consultés en console.

Code Python
```
>>> len.__doc__
'Return the number of items in a container.'
>>> help(len)
Help on built-in function len in module builtins:
```

**Propriété 3: Créer ses propres docstrings**

Code Python
```
def capital_apres_n_annees(capital, taux, nombre_annees) :
    '''
    Renvoie le capital après n années.
    capital : valeur initiale
    taux : taux d'intérêt exprimé en nombre décimal
    nombre_annees : nombre d'années de placement du capital
    '''
    return capital * (1 + taux)**nombre_annees
```

## 3. Programmation défensive: les `asserts`

**Définition 2: Programmation défensive**

La programmation défensive est l'art de prévoir le pire et d'essayer de le détecter avant qu'il ne soit trop tard. De manière bien plus concrète, il est d'usage d'essayer de repérer si des données (souvent des paramètres d'une fonction) sont susceptibles de créer des problèmes, ou sont hors spécification.

</> Code Python

```
1 def racine_carree(x):
2     assert x >= 0, 'un nombre positif ou nul est requis'
3     return x ** 0.5
```

</> Code Python

```
1 >>> racine_carree(-2)
2 Traceback (most recent call last):
3   File "<pyshell>", line 1, in <module>
4   File "/home/gilles/Bureau/exemples_assert.py", line 2, in racine_carree
5     assert x >= 0, 'un nombre positif ou nul est requis'
6 AssertionError: un nombre positif ou nul est requis
```

**Remarque 1: Gestion des asserts**

À ce stade, les `assert` sont donc pour nous juste un moyen rapide de remplacer un test `if ... else` pour détecter des erreurs potentielles.
Ils sont en réalité plus utiles que cela : lors de la conception d'un programme, des assert sont posés pour vérifier l'intégrité du code, mais peuvent être désactivés à tout moment pour en faire un code optimisé (par la commande -o à l'exécution).

## 4. Les tests

Tester une fonction est la première chose que l'on fait (normalement...) lorsqu'on vient de finir de l'écrire.

**Propriété 4: Gestion des tests**

On peut regrouper tous ces tests au sein d'une meme fonction `test_valeur_absolue()`.
On peut remplacer les `if ... else` par des `assert`.

</> Code Python

```
1 >>> valeur_absolue(-3)
2 3
3 >>> valeur_absolue(0)
4 0
```

</> Code Python

```
1  def test_valeur_absolue():
2      if valeur_absolue(-3) == 3 :
3          print("ok")
4      else:
5          print("erreur")
6
7      if valeur_absolue(0) == 0 :
8          print("ok")
9      else:
10         print("erreur")
11
12 def test_valeur_absolue():
13     assert valeur_absolue(-3) == 3
14     assert valeur_absolue(0) == 0
```

**Propriété 5: Le module `doctest` permet d'écrire les tests à l'intérieur de la `docstring` d'une fonction.**

</> Code Python

```
1  import doctest
2
3  def valeur_absolue(n):
4      '''
5      >>> valeur_absolue(-3)
6      3
7      '''
8      if n < 0:
9          n = -n
10     return n
```

</> Code Python

```
1 >>> doctest.testmod()
2 TestResults(failed=0, attempted=1)
```

## 5. Les exceptions en python

Quand le moteur Python rencontre une erreur, il lève une **exception**. Une exception est un objet en python qui est définie par son type. Voici une liste non exhaustive de types d'exception:

| Type d'erreur | Description |
|---|---|
| `SyntaxError` | Le code ne peut pas être exécuté car l'analyseur a repéré une erreur dans la syntaxe de python. Il peut s'agir de deux-point ":" manquant, de parenthèses mal couplées ou d'indentation manquante. L'analyseur qui ne reconnaît plus la syntaxe de python, signale une ligne d'erreur mais attention la faute de syntaxe est souvent localisée avant cette ligne. |
| `ZeroDivisionError` | Erreur de division par zéro. |
| `NameError` | Une variable, une fonction, une classe, invoquée ... n'est pas précédement définie. |
| `TypeError` | L'opérateur n'accepte pas le type d'**opérande** qui lui est donné. |
| `ValueError` | Ici l'opérateur reçoit le bon type d'opérande mais sa valeur est inappropriée. |
| `IOError` | Le fichier n'existe pas. |
| `RecursionError` | La limite de la pile d'appels récursifs a été atteinte. Elle est de 1000 par défaut et peut être modifiée. |
| `IndexError` | L'indice pour une séquence, ou la clé pour un dictionnaire n'existe pas. |
| `AssertionError` | Une assertion est fausse. |
| `KeyboardInterrupt` | L'exécution est stoppée par l'utilisateur (fermeture de fenêtre par exemple). Contrairement aux autres erreurs, il ne s'agit pas d'un objet `Exception`, en python. Par exemple, il ne sera pas capter par l'instruction `except Exception :`, contrairement à toutes les autres. Par contre, il sera capter par l'instruction `except :`. |

**</> Code Python**

```python
try:
    # instructions à exécuter
except """type d'erreur 1""" :
    # instructions à exécuter si ce type d'erreur est levée
except """type d'erreur 2""" :  # Un deuxième except est facultatif
    # instructions à exécuter si ce type d'erreur est levée
else:  # Factultatif
    # instructions à exécuter s'il n'y a pas eu d'erreur
finally:  # Facultatif
    # instructions à exécuter dans tous les cas même si un return existe
    # dans un bloc except
```

**</> Code Python**

```python
import math
def fonction(a, b):
    try:
        y = math.sqrt(a) / b
    except TypeError:
        return f'Les valeurs doivent être des nombres'
    except ValueError:
        return f'On ne prend pas la racine carrée d\'un nombre négatif'
    except ZeroDivisionError:
        return f'Il est impossible de diviser par 0'
    else:
        return y
    finally:
        print('Tache complétée')
```

**</> Code Python**

```
>>> fonction(6, 2)
Tache complétée
1.224744871391589
>>> fonction(6, 0)
Tache complétée
'Il est impossible de diviser par 0'
>>> fonction(-6, 2)
Tache complétée
"On ne prend pas la racine carrée d'un nombre négatif"
>>> fonction('a', 2)
Tache complétée
'Les valeurs doivent être des nombres'
```

# 6. Modularité

**Définition 3: La modularité**

La **modularité** est le fait de décomposer un programme en plusieurs fonctions/modules/fichiers.

- Le découpage en sous-parties indépendantes permet la réutilisation du code pour d'autres projets.
- La structuration d'un projet en plusieurs sous-projets, rend le projet initial plus réalisable, compréhensible et maintenable.
- La modularité est une nécessité pour l'organisation d'un projet et le partage des tâches au sein d'une équipe.

Un module constitue une "brique" logicielle d'un projet, pouvant éventuellement être utilisé par d'autres projets.

En python, les modules sont importés grâce au mot clé import proposant plusieurs syntaxes:

Code Python
```python
# Syntaxe 1
from math import *
print(pi)
```

Code Python
```python
# Syntaxe 2
from math import pi
print(pi)
```

Code Python
```python
# Syntaxe 3
import math
print(math.pi)
```

Code Python
```python
# Syntaxe 4
import math as m
print(m.pi)
```

Pour créer vos propres modules, il suffit de créer un fichier python `mon_module.py` à la racine de votre projet et de l'importer avec les mêmes syntaxes proposées par l'instruction `import`.

Code Python
```python
from mon_module import *
from mon_module import ma_fonction
import mon_module
import mon_module as mm
```

Il existe un grand nombre de **modules natifs** au moteur Python. La liste exhaustive est disponible dans la Doc Python en ligne. Parmi eux, en voici quelques-uns particulièrement utilisés:

- tkinter: interface graphique
- math: fonction et constantes mathématiques
- random: génération de nombres aléatoires
- time: fonctions liées au temps
- os: interaction avec le système d'exploitation, gestion des fichiers
- turtle: affichage graphique
- sqlite3: interaction avec une BD
- pip: gestion et installation de modules supplémentaires

# LP02 – Récursivité

**Définition 1: Récursivité**

Un algorithme (ou fonction) récursif est un algorithme (ou fonction) qui fait appel à lui-même.
La principe de la programmation récursive est d'utiliser:

- le cas de base (ou trivial). C'est la condition d'arrêt. (en général c'est une condition facile à trouver car c'est le cas le plus simple).
- une relation de récurrence entre le cas à un instant donné et ce même cas à l'instant suivant.

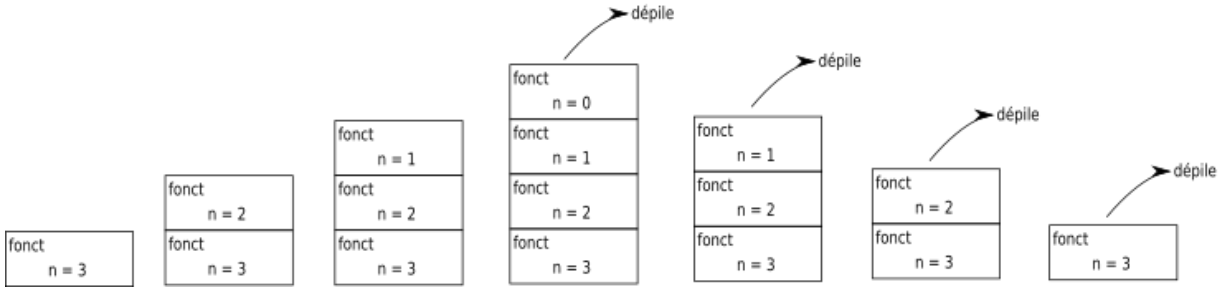
**Exemple 1: Les suites récurrentes**

Soit la suite $(u_n)$ définie pour tout entier naturel par:

$$\begin{cases} u_0 = 2 \\ u_{n+1} = 2u_n + 1 \end{cases}$$

</> Code Python

```
def suite(n):
    if n == 0:
        return 2
    else:
        return 2 * suite(n-1) + 1
```




| appel | n | IF | appel récursif | valeur renvoyée |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |

suite(3)
- suite(2)
  - suite(1)
    - suite(0) → 2
    - Calcul: $2 \times 2 + 1 = 5$
  - Calcul: $2 \times 5 + 1 = 11$
- Calcul: $2 \times 11 + 1 = 23$

**Exemple 2: La puissance récursive**

Soit la fonction définie par:

$$\begin{cases} a^0 = 1 \\ a^{n+1} = a \times a^n \text{ si } n \neq 0 \end{cases}$$

</> Code Python

```
def puissance_rec(a, n):
    if n == 0:
        return 1
    else:
        return a * puissance_rec(a, n-1)
```

**Exemple 3: La factorielle récursive**

Soit la fonction définie par:

$$\begin{cases} 0\,! = 1 \\ (n+1)! = (n+1) \times n! \text{ si } n \neq 0 \end{cases}$$

</> Code Python

```
def facto_rec(a, n):
    if n == 0:
        return 1
    else:
        return n * facto_rec(n-1)
```

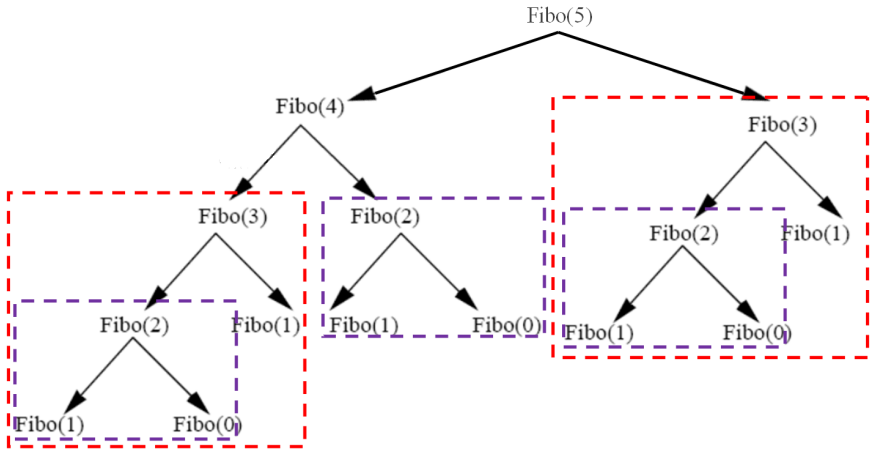
**Exemple 4: Suite de Fibonacci récursive**

$$\begin{cases} F(0) = 0, \quad F(1) = 1 \\ F(n+2) = F(n+1) + F(n) \end{cases}$$

</> Code Python

```
def fibo_rec(n):
    if n == 0 or n == 1
        return n
    else:
        return fibo_rec(n-1) + fibo_rec(n-2)
```

Attention, la récursivité n'est pas toujours un gain de complexité:




```
1 -> fibo(5)
├── 2 -> fibo(4)
│   ├── 3 -> fibo(3)
│   │   ├── 4 -> fibo(2)
│   │   │   ├── 5 -> fibo(1)
│   │   │   └── 6 -> fibo(0)
│   │   └── 7 -> fibo(1)
│   └── 8 -> fibo(2)
│       ├── 9 -> fibo(1)
│       └── 10 -> fibo(0)
└── 11 -> fibo(3)
    ├── 12 -> fibo(2)
    │   ├── 13 -> fibo(1)
    │   └── 14 -> fibo(0)
    └── 15 -> fibo(1)
```

**Exemple 5: D'autres exemples**

Somme des éléments dans une liste:

**Code Python**

```python
def somme(L:list)->int:
    if L == []:
        return 0
    return L.pop() + somme(L)
```

Recherche de la valeur maximale dans une liste

**Code Python**

```python
def maxi(L:list):
    if len(L)==1 or L[0]>maxi(L[1:]):
        return L[0]
    return maxi(L[1:])
```

Recherche la présence d'un élément dans une liste

**Code Python**

```python
def rechecher(L:list, val)->bool:
    if L == []:
        return False
    if L[0] == val:
        return True
    return rechercher(L[1:], val)
```

Vérifie si une chaine de caractère est un palindrome

**Code Python**

```python
def palidrome(mot:str)->bool:
    if len(mot) <= 1:
        return True
    if mot[0] != mot[-1]:
        return False
    else:
        return palindrome(mot[1:-1])
```

Conversion décimal - binaire

**Code Python**

```python
def dec2bin(n:int)->str:
    if n == 0 or n == 1:
        return str(n)
    return dec2bin(n//2) + str(n%2)
```

Conversion binaire décimale

**Code Python**

```python
def bin2dec(val:str, i = 0)->int:
    if len(val) == 1:
        return int(val)
    return int(val[-1])*2**i +\
                bin2dec(val[:-1],i+1)
```

# LP03 – Calculabilité – Décidabilité

## 1. Décidabilité

**Définition 1: Décidabilité**

La **décidabilité** est une notion utilisée en logique. Dans le cadre d'une théorie basée sur un certain nombre d'axiomes (c'est-à-dire de propositions non démontrées) et de leurs conséquences, il s'agit de démontrer qu'une proposition est vraie ou fausse. Si cette démonstration est possible, on dit que la proposition est décidable. Sinon, on dit que la proposition est indécidable.

En algorithmique, un problème de décision est une question à laquelle on répond par **oui** ou par **non**. On dit qu'un problème est **décidable** s'il existe un algorithme, c'est-à-dire une méthode qui se termine en un nombre fini d'étapes, qui permet de répondre par oui ou par non à la question posée par le problème. Sinon on dit que le problème est indécidable.

**Exemple 1: Fonction est_pair**

Est-il possible d'écrire une fonction prenant comme argument un entier $n$ et renvoyant 0 si le nombre $n$ est pair et 1 sinon?

Il s'agit bien d'un **problème de décision** et celui est bien entendu décidable. Notons que nous créons une fonction **calculable** pour résoudre le problème.

**David Hilbert et Kurt Godel**

- David Hilbert a posé en 1928 la question en logique mathématique qu'on appelle le problème de la décision, en allemand "Entscheidungsproblem". Peut-on déterminer par un algorithme si un énoncé quelconque est vrai ou faux, si c'est un théorème?
- Kurt Gödel a démontré en 1931 qu'il existe des propriétés mathématiques non décidables dans n'importe quel système définissant l'arithmétique.
- Alan Turing a lui aussi répondu négativement à la question de Hilbert en utilisant l'indécidabilité d'un problème nommé "le problème de l'arrêt".

**Programme comme paramètre d'un programme**

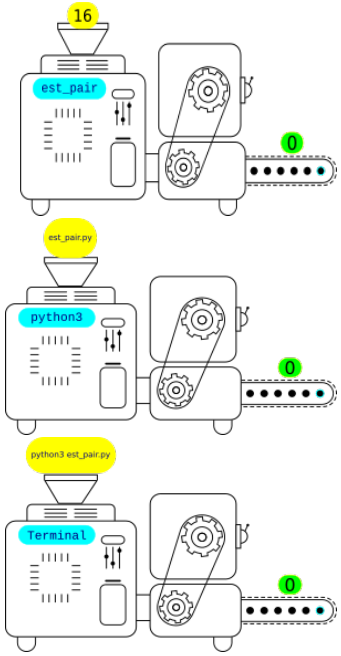
On considère la fonction `est_pair` précédente:
Ce programme prend en argument un paramètre n entier et renvoie 1 ou 0 selon la parité de n.

Code Python

```python
def est_pair(n: int) -> int:
    return n%2
```

Une représentation imagée serait celle d'une machine:

- il y a une entrée, dans laquelle on introduit par exemple le nombre 16
- la machine effectue des opérations sur l'entrée, ici un modulo 2 ;
- elle envoie dans la sortie le résultat, ici 0.




Enregistrons ensuite le code suivant dans un fichier `est_pair.py`.
Pour exécuter le code, nous pouvons nous rendre dans un terminal et taper le code `python3 est_pair.py`

En exécutant cette ligne, on fait appel au programme `python3`, qui prend comme paramètre le fichier `est_pair.py`. Dans ce cadre, `est_pair.py` est un ensemble de caractères qui contient les instructions que le programme `python3` va interpréter, pour écrire en sortie la valeur 0.

Mais en poussant plus loin la réflexion, le terminal est lui-même un programme, qui lit en entrée la chaine de caractères python3 `est_pair.py`, l'interprète puis exécute les instructions demandées, pour au final afficher 0.

On pourrait continuer ainsi en remontant au système d'exploitation, puis au bootloader.

**Propriété 1: Programme et données**

Un programme informatique peut parfaitement être observé comme une donnée pouvant être reçue comme paramètre par un autre programme.

## 2. Le problème de l'arret

Dans cet exemple, il est facile de voir les conditions d'arrêt de ce programme.

Code Python

```python
def countdown(n : int) -> None :
    while n != 0 :
        print(n)
        n -= 1
    print("Go !")
```

**Propriété 2: Problème de l'arrêt**

Est-il possible d'automatiser cette analyse en créant un programme qui prédira l'arrêt ou non d'un autre programme qui lui sera passé en paramètre?

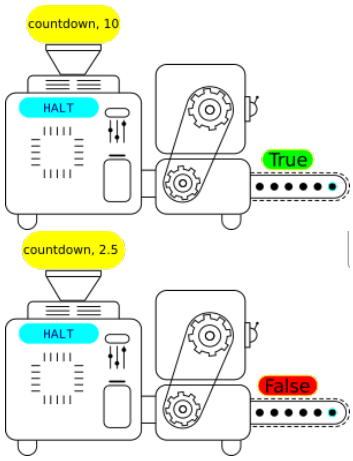
**Exemple 2: Une fonction qui calcule l'arrêt**

Imaginons que nous ayons une fonction `HALT` prenant en paramètre:

- `prog` le code source d'un programme;
- `x` un paramètre (ou une liste de paramètres) pour le programme `prog`

Cette fonction renverrait:

- `True` si `prog(x)` s'arrête ;
- `False` si `prog(x)` ne s'arrête pas.




**Exemple 3: Une nouvelle construction**

On constate que ce programme teste si `prog(prog)` s'arrête. Le programme à alors deux possibilités:

- si `HALT(prog, prog)` est `True`, ce qui signifie que `prog(prog)` s'arrete, alors `sym(prog)` rentre dans une boucle infinie;
- si `HALT(prog, prog)` est `False`, ce qui signifie que `prog(prog)` ne s'arrete pas, alors `sym(prog)` renvoie 1.

Code Python

```python
def sym(prog) :
    if HALT(prog, prog):
        while True:
            print("Ca boucle!")
    else:
        return 1
```

Que se passe-t-il si on applique le code source de `sym` à elle-meme, c'est-à-dire qu'on appelle `sym(sym)`:

- Si `HALT(sym, sym)` est `True`, alors l'appel `sym(sym)` rentre dans la boucle infinie. Pourtant, pour obtenir ce comportement, il faut que `sym(sym)` s'arrete. Ce qui donne une contradiction.
- Si `HALT(sym, sym)` est `False`, alors l'appel `sym(sym)` renvoie 1. Pourtant, pour obtenir ce comportement, il faut que `sym(sym)` ne s'arrête pas. Ce qui donne une contradiction.

**Propriété 3: Conclusion**

Le programme `HALT`, sensé prédire si un programme `prog` appliqué à une donnée `x` ne peut donc pas exister. Le problème de l'arrêt est donc indécidable.

Cette question, appelée "problème de la décision", ou "Entscheidungsproblem" en allemand, est définitivement tranchée par le problème de l'arrêt: un tel théorème ne peut pas exister, puisque par exemple, aucun algorithme ne peut répondre oui ou non à la question "ce programme va-t-il s'arrêter?".

# 3. Calculabilité

**Définition 2: Décidabilité et calculabilité**

Le problème de l'arrêt est dit indécidable car la fonction qui le résout (programme `halt`) n'est pas calculable.

La notion de calculabilité crée un lien entre les mathématiques (Thèse de Church) et l'informatique (vision de Turing).

Le calcul mathématique peut se réduire à une succession d'opérations élémentaires. Les nombres calculables sont les nombres qui sont générables en un nombre fini d'opérations élémentaires.

Certaines fonction peuvent être calculables et d'autres non: c'est le cas par exemple de la fonction du problème de l'arrêt.

**Propriété 4: Un classement des problèmes**

1. Problèmes ayant une solution en temps polynomial:
   - deux entiers $n$ et $p$ sont-ils premiers entre eux?
   - un programme en python est-il syntaxiquement correct?
   - un graphe est-il connexe?
   - problème du tri dans une liste.

   On note `P` l'ensemble des problèmes résolus en un temps polynomial.
2. Problèmes ayant une solution mais pas en temps polynomial:
   - Une grille de Sudoku a-t-elle une solution?
   - Un graphe donné peut-il être coloré avec trois couleurs sans que deux sommets voisins aient la même couleur?
   - un graphe est-il connexe?
   - Étant donné un entier $d$, des villes et les distances entre les villes, existe-t-il un trajet passant par toutes les villes de longueur inférieur à $d$ ? (problème du voyageur de commerce, TSP).

Entre ces deux grandes catégories, on trouve la catégorie des problèmes `NP` (Non déterministe Polynomial). Il s'agit des problèmes pour lesquels la vérification d'une solution est en temps polynomial mais la recherche d'une solution est à priori en temps exponentiel.

On se demande s'il existe pour ces problèmes des solutions en temps polynomial. C'est une des plus grandes questions ouvertes en informatique. Est-ce que `P = NP`?

**Exercice 1:** Sujet de BAC – 2024 Métropole Septembre – J2 – Exercice 1

# LP04 – Paradigmes de programmation

**Propriété 1: Les différents paradigmes de programmation**

Un **paradigme de programmation** est une approche particulière ou un style de programmation. Chaque paradigme a ses propres règles, conventions et meilleures pratiques.

**Propriété 2: Paradigme impératif**

La programmation **impérative** repose sur des notions qui vous sont familières:

- la séquence d'instructions (les instructions d'un programme s'exécutent l'une après l'autre)
- l'affectation (on attribue une valeur à une variable, par exemple: `a = 5`)
- l'instruction conditionnelle (`if / else`)
- la boucle (`while et for`)

Dans le paradigme impératif, les données sont stockées dans des variables et le programme s'organise comme une séquence d'instructions.

Les unités de code réutilisables peuvent être stockées dans des fonctions (programmation structurée).

**Propriété 3: Paradigme objet**

- Le **paradigme objet organise** les données en une collection d'objets dont l'état interne (attributs) peut être modifié à l'aide de méthodes.
- Les objets sont instanciés à partir de classes qui étendent la notion de type du paradigme impératif.
- L'encapsulation permet d'éviter que les attributs soient modifiés depuis l'extérieur de la classe.

**Propriété 4: Paradigme fonctionnel**

Le **paradigme fonctionnel** organise un programme comme un enchainement d'évaluations de fonctions, chaque résultat produit en sortie d'une fonction étant pris en entrée de la fonction suivante.

- La valeur d'une variable ne change pas. Les structures de données sont immuables. Cela permet d'empêcher les effets de bord.
- Il n'existe donc pas d'instructions comme l'affectation qui peuvent modifier l'état du programme. Le calcul repose sur l'évaluation d'expressions, qui ont une valeur, et de fonctions, qui associent à une valeur, une autre valeur.
- Les fonctions sont des valeurs comme autres. Une fonction peut être argument d'une autre fonction, valeur de retour d'une autre fonction, stockée dans une structure de données.
- Une fonction peut donc s'appliquer à d'autres fonctions, on parle de fonction d'ordre supérieur: les analogies mathématiques sont la composition de fonction, la dérivation, l'intégration ...
- Les structures d'itération comme les boucles du paradigme impératif sont remplacées par la récursion.
- Les fonctions sont des fonctions pures c'est-à-dire qu'elles ne provoquent pas d'effets de bord lors de leur évaluation et que pour des entrées fixées, elles donnent toujours le même résultat en sortie. Cette propriété garantit la transparence référentielle c'est-à-dire que tout appel de fonction peut être remplacé par la valeur de son évaluation sans modifier le programme. Ceci ne serait pas garanti avec une fonction impure dont l'évaluation pourrait s'accompagner d'effets de bord en plus du calcul du résultat.

# 1. Le paradigme fonctionnel

**Exercice 1:** Effets de bord

1. Afficher la liste `une_liste`.
2. Utiliser les deux fonctions sur `une_liste`
3. Observer le résultat
4. Expliquer

**Code Python**

```
def ajout_1(ma_liste, i):
    ma_liste.append(i)
def ajout_2(ma_liste, i):
    return ma_liste + [i]
une_liste = [4, 7, 3]
```

**Propriété 5: Les fonctions pures**

- Une **fonction pure** est une fonction qui ne modifie rien;
- Les modifications qu'une fonction peut effectuer sur l'état du système sont appelées effets de bord. Un affichage à l'écran est un exemple d'effet de bord.

**Propriété 6: Fonctions d'ordre supérieur**

Les fonctions sont des objets de première classe, ce qui signifie qu'elles sont manipulables aussi simplement que les types de base.

Une fonction peut prendre des fonctions comme paramètres ou renvoyer une fonction comme résultat.

**Exemple 1: Utilisation de `sorted`**

**Code Python**

```
def clef_note(mon_tuple):
    return (mon_tuple[0])
def clef_nom(mon_tuple):
    return (mon_tuple[1])

tab = [(16, 'MARIE') , (16, 'ISMAEL'), (12, 'ANNE'), (17, 'SARAH')]
print("ordre décroissant des notes: ", sorted(tab, key=clef_note, reverse=True))
print("ordre alphabétique des noms: ", sorted(tab, key=clef_nom, reverse=False))
```

La fonction `sorted` est une fonction d'**ordre supérieur**, qui prend en paramètre une fonction, comme ici `clef_note` ou `clef_nom`.

**Exemple 2: Fonctions anonymes et opérateur `lambda`**

**Code Python**

```
def double(x):
    return 2 * x
```

**Code Python**

```
# Remplacable par
double = lambda x: 2 * x
```

**Code Python**

```
print(sorted(tab, key=lambda mon_tuple:mon_tuple[0], reverse=True))
print(sorted(tab, key=lambda mon_tuple:mon_tuple[1], reverse=False))
```

**Code Python**

```
# Deux fonctions en paramètres
def somme_fonctions(f, g, x):
    return f(x) + g(x)
def f1(x):
    return x**2
def f2(x):
    return 2*x
```

**Code Python**

```
# Fonction renvoyant une fonction
def somme_fonctions(f, g):
    return lambda x: f(x) + g(x)
def f1(x):
    return x**2
def f2(x):
    return 2*x
k = somme_fonctions(f1, f2)
```

**Exemple 3: Fonction qui renvoie une fonction**

**Code Python**

```
affine = lambda a, b: lambda x: a*x + b
f1 = affine(3, -2)
print(f1(6))
```

### Exemple 4: Fonction `map`

La fonction map est une fonction qui permet d'appliquer un traitement à tous les éléments d'un itérable. Cette fonction ne modifie pas l'objet de départ : elle renvoie un objet (itérable) encapsulant le résultat.

**Code Python**

```python
def carre(x):
  return x ** 2

nombres = (1, 2, 3, 4, 5)
resultat = map(carre, nombres)
```

**Code Python**

```
>>> print(f"resultat : {resultat}")  # Un objet itérable
resultat : <map object at 0x106cc38>
>>> print(f"type(resultat) : {type(resultat)}")
type(resultat) : <class 'map'>
>>> print(f"list(resultat) : {list(resultat)}")
list(resultat) : [1, 4, 9, 16, 25]
```

### Exemple 5: Fonction `filter`

La fonction `filter` est un autre exemple de fonction d'ordre supérieur s'appliquant à des objets itérables. Elle prend en premier paramètre une fonction à valeur booléenne appelée filtre, et un objet itérable en deuxième paramètre. En résultat, elle renvoie un itérable ne contenant que les valeurs de la liste pour lesquels le filtre renvoie la valeur `True`.

**Code Python**

```python
def est_pair(n):
  return n % 2 == 0

nombres = (1, 2, 3, 4, 5)
resultat = filter(est_pair, nombres)
```

**Code Python**

```
>>> print(f"resultat : {resultat}")  # Un objet itérable
resultat : <filter object at 0xdd5498>
>>> print(f"type(resultat) : {type(resultat)}")
type(resultat) : <class 'filter'>
>>> print(f"list(resultat) : {list(resultat)}")
list(resultat) : [2, 4]
```

On aurait pu utiliser une fonction anonyme.

**Code Python**

```python
nombres = (1, 2, 3, 4, 5)
resultat = filter(lambda x: x % 2 == 0, nombres)
```