CPGE TSI

Synthèse d'informatique

Python 3

Jean Decroocq

0. Sommaire

1. Fondements

1.1. Variables et types

– Une variable ne constitue pas un conteneur physique, mais une référence (une étiquette) vers un objet stocké en mémoire. L'affectation, réalisée via le signe `=`, lie un identifiant à une valeur. Le typage est dynamique : la nature de la variable est déterminée par la valeur qu'elle reçoit au moment de l'exécution.

– Les nombres entiers (`int`) possèdent une précision arbitraire en Python, ce qui signifie qu'ils ne sont limités que par la mémoire de la machine (pas de dépassement de capacité standard). Ils supportent les opérations classiques ainsi que la division euclidienne (`//`), le modulo (`%`) et l'exponentiation (`**`).

– Les nombres flottants (`float`) représentent les réels via la norme IEEE 754. Cette représentation est intrinsèquement approximative pour certaines valeurs décimales (ex : `0.1 + 0.2 != 0.3`). Ils peuvent s'écrire en notation scientifique : `1.5e4` pour $1,5 \times 10^4$.

– Les booléens (`bool`) ne peuvent prendre que deux états : `True` ou `False`. Ils sont le résultat d'opérations de comparaison (`==`, `!=`, `<`, `>=`) et se combinent à l'aide des connecteurs logiques `not`, `and` et `or`.

– Les chaînes de caractères (`str`) sont des séquences immuables. On ne peut modifier un caractère directement (`s[0] = 'a'` lève une erreur). La concaténation s'effectue avec `+` et la répétition avec `*`.

– Les listes (`list`) sont des séquences mutables, ordonnées et hétérogènes, définies entre crochets. L'accès aux éléments se fait par un indice entier positif (depuis 0) ou négatif (depuis la fin, -1 étant le dernier élément).

```
L = [10, 20, 30]
L[0] = 5          # Modification en place : L vaut [5, 20, 30]
dernier = L[-1]   # Accede au dernier element (30)
```

– Il est possible de convertir explicitement une valeur d'un type vers un autre (casting).
- `int(x)` : Convertit en entier.
  Depuis un flottant : réalise une troncature de la partie décimale vers zéro (`int(3.9) -> 3`, `int(-2.7) -> -2`).
  Depuis une chaîne : la chaîne doit représenter strictement un entier (`int("42") -> 42`), sinon une `ValueError` est levée (`int("3.5")` échoue).
- `float(x)` : Convertit en flottant. Utile pour forcer la précision décimale dans des calculs (`float("1e-4") -> 0.0001`).
- `str(x)` : Transforme n'importe quel objet en sa représentation textuelle, ce qui est indispensable pour l'affichage ou la concaténation (`"Val=" + str(10)`).
- `bool(x)` : Convertit en booléen. En Python, la plupart des valeurs sont considérées comme vraies (`True`), sauf les valeurs "vides" ou nulles qui valent `False` : le nombre `0`, `0.0`, la chaîne vide `""`, les listes vides `[]`, et la valeur `None`.

1.2. Fonctions

– Une fonction encapsule un bloc d'instructions afin de le rendre réutilisable. Elle est définie par le mot-clé `def`, suivi de son nom et de ses paramètres entre parenthèses. Le bloc d'instructions constituant le corps de la fonction doit impérativement être indenté (décalé vers la droite).

– L'instruction `return` interrompt l'exécution de la fonction et renvoie la valeur spécifiée à l'endroit où la fonction a été appelée. Une fonction sans instruction `return` (ou avec un `return` vide) renvoie implicitement la valeur `None`.

```
def f(x, a=1): #Exemple avec un parametre optionnel a valant 1 par defaut
    y = x ** 2 + a
    return y
```

– Les variables définies à l'intérieur d'une fonction ont une portée locale : elles n'existent que durant l'exécution de celle-ci et sont détruites ensuite. Elles ne peuvent pas modifier directement une variable globale immuable, sauf usage explicite (et déconseillé) du mot-clé `global`.

1.3. Algorithmique et programmation

– L'instruction conditionnelle dirige le flot d'exécution. Si la condition du `if` est fausse, le programme teste les éventuels `elif` successifs. Si aucune condition n'est vérifiée, le bloc `else` est exécuté.

```
if x > 0:
    signe = 1
elif x < 0:
    signe = -1
else:
    signe = 0
```

– La boucle bornée `for` est utilisée pour parcourir un itérable (liste, chaîne, tuple) ou réaliser un nombre d'itérations déterminé. On l'associe souvent au générateur `range(start, stop, step)`. Notez bien que la borne supérieure `stop` est stricte (exclue).

- `range(5)` génère : 0, 1, 2, 3, 4.
- `range(1, 5)` génère : 1, 2, 3, 4.
- `range(0, 10, 2)` génère : 0, 2, 4, 6, 8.

– La boucle non bornée `while` répète un bloc d'instructions tant qu'une condition booléenne reste vraie. Il est crucial de s'assurer que l'état du programme évolue vers la terminaison de la boucle pour éviter une boucle infinie.

```
n = 0
while n < 10:
    print(n)
    n = n + 1  # Indispensable pour que la condition devienne fausse
```

2. Tableaux Numpy

2.1. Pour commencer

- Le module `numpy` est essentiel pour le calcul scientifique nécessitant la manipulation de grandes quantités de données. Il repose sur trois principes d'efficacité : le stockage sous forme de tableaux `ndarray`, la limitation des copies mémoire, et l'utilisation de fonctions vectorialisées pour éviter les boucles.
- Ces optimisations imposent des contraintes : les tableaux sont constitués d'éléments de même type (homogènes, souvent `float` ou `int64`) et leur taille est fixée à la création (on ne peut pas augmenter la taille comme pour une liste).
- Le chargement du module se fait traditionnellement via l'instruction : `import numpy as np`.

2.2. Les tableaux numpy

- La création basique se fait via `np.array()` à partir d'une liste. L'attribut `.dtype` indique le type commun (ex : `int64`), différent des listes Python.
- Pour les vecteurs (1D), on utilise `np.arange(start, stop, step)` qui accepte des pas flottants, ou `np.linspace(start, stop, num)` pour obtenir `num` points équitablement répartis (bornes incluses).
- Des matrices spéciales sont prédéfinies : `np.ones()`, `np.zeros()`, `np.eye()` (identité) et `np.diag()`. Le module `np.random` permet de générer des tableaux aléatoires (uniformes, binomiaux, géométriques...). La lecture de fichiers CSV se fait avec `np.genfromtxt`.
- Un tableau possède des attributs clés : `dtype`, `size` (nombre d'éléments), `shape` (tuple des dimensions) et `ndim`. L'attribut `shape` est mutable : on peut redimensionner un tableau via la méthode `reshape()`.
- Le stockage en mémoire est contigu. L'attribut `dtype` est immuable pour garantir la cohérence de l'espace mémoire.
- Le *slicing* (tranchage) s'étend à plusieurs dimensions (ex : `T[ :, 0]` pour la première colonne). **Attention** : le slicing renvoie une **vue** et non une copie. Modifier une tranche modifie le tableau d'origine. Pour copier, il faut utiliser `.copy()`.
- Les masques (indexation booléenne) permettent de filtrer un tableau. Si `b` est un tableau de booléens de même format que `a`, alors `a[b]` extrait les éléments où `b` est `True`.

2.3. Opérations sur les tableaux numpy

- Les opérations arithmétiques usuelles (`+`, `*`, `**`...) s'appliquent **terme à terme** (élément par élément). C'est la vectorisation.
- Il ne faut pas faire de copies inutiles de grands tableaux. Les opérations se font souvent au travers de vues.
- Les fonctions universelles s'appliquent aussi terme à terme. On peut vectoriser une fonction personnelle avec `np.vectorize`.
- Des méthodes statistiques sont disponibles : `max`, `min`, `sum`, `prod`, `mean`, `var`, `std`.

– Pour l'algèbre linéaire, le produit matriciel se fait avec `np.dot(a, b)`. Le produit scalaire canonique utilise `np.vdot` et le produit vectoriel `np.cross`. La transposée s'obtient avec `.T` ou `.transpose()`.

– Le sous-module `np.linalg` fournit des outils avancés : `solve` (résolution de système), `inv` (inverse), `det` (déterminant), `norm` (norme) et `eig` (éléments propres).

– La classe `Polynomial` (du module `numpy.polynomial`) permet de manipuler formellement des polynômes (racines, dérivées, primitives) définis par leurs coefficients.

3. Compléments sur les listes Python et chaînes de caractères

3.1. Tableaux et listes Python

– Les listes Python (`list`) sont des structures de données mutables, ordonnées et hétérogènes. Bien que souvent appelées "listes", elles sont implémentées sous forme de tableaux dynamiques (vecteurs) contenant des références vers les objets. Cela garantit un accès à n'importe quel élément en temps constant $O(1)$.

– La construction d'une liste peut se faire par extension (écriture littérale) ou, de manière plus élégante et performante, par compréhension. Cette dernière méthode permet de générer une liste en appliquant une expression à chaque élément d'un itérable, éventuellement filtré par une condition.

```
1 L1 = [0, 1, 4, 9, 16] # Par extension
2 L2 = [k**2 for k in range(5)] # Par comprehension (identique a L1)
3 L3 = [x for x in L1 if x % 2 == 0] # Avec filtrage (nombres pairs)
```

– Le mécanisme de tranchage (*slicing*) permet d'extraire une sous-partie de la liste en créant une **nouvelle** liste (copie). La syntaxe générale est `L[debut :fin :pas]`.

- `L[i :j]` : sélectionne les éléments de l'indice i inclus à j exclu.
- `L[i :]` : sélectionne de l'indice i jusqu'à la fin.
- `L[ :j]` : sélectionne du début jusqu'à l'indice j exclu.
- `L[ : :-1]` : crée une copie renversée de la liste.

– Les listes étant mutables, elles disposent de méthodes agissant *in-place* (modifiant l'objet sans le renvoyer). Les plus courantes sont :

- `L.append(x)` : ajoute l'élément x à la fin de la liste (opération en temps constant $O(1)$).
- `L.pop()` : supprime et renvoie le dernier élément (également en $O(1)$). `L.pop(i)` supprime l'élément à l'indice i.
- `L.sort()` : trie la liste en place (algorithme Timsort).
- `L.reverse()` : inverse l'ordre des éléments en place.

D'autres opérations existent comme la concaténation (`L1 + L2`) ou la mesure de longueur (`len(L)`).

– Un point de vigilance majeur concerne l'aliasing. L'instruction `L2 = L1` ne copie pas la liste, mais crée une seconde référence vers le même objet en mémoire. Pour obtenir une copie indépendante (superficielle), il faut utiliser le tranchage complet `L[ :]` ou la méthode `L.copy()`.

3.2. Chaînes de caractères

– Les chaînes de caractères (`str`) sont des séquences ordonnées de caractères Unicode. Contrairement aux listes, elles sont **immuables** : il est impossible de modifier un caractère par affectation directe (`ch[0] = 'a'` lève une erreur `TypeError`). Toute opération de modification renvoie nécessairement une nouvelle chaîne.

– La syntaxe de tranchage (*slicing*) et les fonctions universelles de séquences (`len()`, `in`) fonctionnent exactement comme pour les listes. La conversion d'un objet en chaîne se fait via le constructeur `str()`.

– Le langage propose des méthodes spécifiques pour la manipulation de texte :
  - `s.find(motif)` : renvoie l'indice de la première occurrence du motif (ou -1 si absent).
  - `s.count(motif)` : compte le nombre d'occurrences du motif.
  - `s.strip()` : retire les espaces (et caractères invisibles) en début et fin de chaîne.
  - `s.replace(old, new)` : remplace toutes les occurrences de la sous-chaîne `old` par `new`.
– Deux méthodes sont essentielles pour passer du type `str` au type `list` et inversement :
  - `sep.join(liste)` : concatène les éléments d'une liste de chaînes en les séparant par la chaîne `sep`.
  - `s.split(sep)` : découpe la chaîne `s` en une liste de sous-chaînes, en utilisant `sep` comme délimiteur.

```
phrase = "Sciences du Numerique"
mots = phrase.split(" ")  # Renvoie ['Sciences', 'du', 'Numerique']
reconst = "-".join(mots)  # Renvoie "Sciences-du-Numerique"
```

4. Dictionnaires

4.1. Création et modification d'un dictionnaire

- Le type `dict` (dictionnaire ou table d'association) permet de stocker des couples clé-valeur. Contrairement aux listes indexées par des entiers, les dictionnaires sont indexés par des clés appartenant à un ensemble C, associées à des valeurs d'un ensemble V.
- Une contrainte fondamentale concerne les clés : elles doivent être de type **hachable** (immuable), comme les `int`, `float`, `str` ou `tuple`. Les valeurs, en revanche, peuvent être de n'importe quel type et sont mutables.
- L'intérêt majeur de cette structure (implémentée par table de hachage) est l'efficacité : l'accès à une valeur via sa clé se fait en temps constant moyen $O(1)$, quelle que soit la taille du dictionnaire.
- La création s'effectue via des accolades `{}` ou le constructeur `dict()`. On peut définir un dictionnaire par extension ou par compréhension.

```
d1 = {} # Dictionnaire vide
d2 = {'argent': 'silver', 'or': 'gold'} # Par extension
d3 = {x: x**2 for x in range(5)} # Par comprehension
# d3 vaut {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

- L'ajout ou la modification d'une entrée utilise la syntaxe d'affectation `d[cle] = valeur`. Si la clé existe déjà, l'ancienne valeur est écrasée (unicité des clés). Si elle n'existe pas, une nouvelle entrée est créée.

4.2. Commandes principales

- La fonction `len(d)` renvoie le nombre de paires clé-valeur stockées.
- Le test d'appartenance `k in d` vérifie si la clé `k` est présente dans le dictionnaire (opération très rapide). Attention, cela ne teste pas la présence des valeurs.
- Pour parcourir un dictionnaire, on dispose de trois méthodes renvoyant des *vues* (objets itérables dynamiques) :
  - `d.keys()` : itère sur les clés (comportement par défaut d'une boucle `for k in d`).
  - `d.values()` : itère sur les valeurs.
  - `d.items()` : itère sur les couples (clé, valeur).

```
d = {'pomme': 2, 'poire': 5}
for k, v in d.items():
    print(k, "->", v) # Affiche "pomme -> 2" etc.
```

- Pour récupérer une valeur, deux approches existent :
  - `d[k]` : renvoie la valeur associée à `k`, mais lève une erreur `KeyError` si la clé est absente.
  - `d.get(k)` : renvoie la valeur si elle existe, ou `None` (ou une valeur par défaut spécifiée) sinon, sans provoquer d'erreur.
- La suppression d'une entrée se fait via la méthode `d.pop(k)`, qui supprime la clé `k` et renvoie la valeur associée. Si la clé n'existe pas, une erreur est levée.

5. Modules et lecture de fichiers

5.1. Les modules

- Un module est une bibliothèque contenant un ensemble de fonctions et de variables prédéfinies, permettant d'étendre les fonctionnalités natives de Python. Des modules standards comme `math`, `random` ou `time` sont disponibles immédiatement, tandis que d'autres comme `numpy` ou `matplotlib` doivent être installés.
- L'importation d'un module peut se faire de plusieurs manières. La méthode recommandée est l'import global, éventuellement avec un alias pour alléger l'écriture.

```
import math
y = math.cos(math.pi)

import numpy as np       # Alias standard
T = np.array([1, 2, 3])
```

- Il est possible d'importer spécifiquement certaines fonctions dans l'espace de nommage courant avec `from ... import ...`. En revanche, l'importation totale via `from module import *` est fortement déconseillée car elle pollue l'espace de noms et peut créer des conflits (aliasing) difficiles à déboguer.
- La fonction `help(module)` ou `help(fonction)` permet d'accéder à la documentation intégrée, décrivant les paramètres attendus et le type de retour.

5.2. Lecture d'un fichier texte de données

- La gestion de l'encodage est primordiale lors de l'ouverture d'un fichier texte. Bien que la norme actuelle soit l'UTF-8 (compatible Unicode), des formats hérités (ASCII) ou propriétaires (Windows CP1252) persistent. Pour éviter les erreurs d'interprétation des accents, il convient de préciser systématiquement l'argument `encoding="utf-8"`.
- Le format CSV (*Comma Separated Values*) est un standard de fichier texte où les données sont séparées par un délimiteur (virgule, point-virgule ou tabulation).
- La lecture "naïve" en Python pur s'effectue avec l'instruction `with open(...) as f :`, qui garantit la fermeture du fichier après lecture. On utilise ensuite `f.readlines()` pour obtenir une liste de chaînes, qu'il faut nettoyer (`.strip()`) et découper (`.split()`).
- Le module `csv` simplifie cette démarche grâce à l'objet `csv.reader`, qui gère automatiquement le découpage selon un délimiteur donné. Cependant, les données lues restent des chaînes de caractères qu'il faut convertir explicitement (en `float` ou `int`).
- La méthode avancée (et recommandée pour le calcul scientifique) repose sur la fonction `np.loadtxt` de Numpy. Elle charge directement les données dans un tableau, gère la conversion de type, et permet de sélectionner les colonnes ou d'ignorer les en-têtes.

```
import numpy as np

# Chargement direct dans des tableaux (unpack=True)
temps, vitesse = np.loadtxt(
    "data.csv",
    delimiter=";",       # separateur
    skiprows=1,          # ignore la 1ere ligne (en-tetes)
    usecols=(0, 2),      # selectionne colonnes 0 et 2
    encoding="utf-8",
```

```
    unpack=True          # transpose pour separer les vecteurs
)
```

5.3. Tracé des courbes correspondantes

– La bibliothèque `matplotlib.pyplot` (importée usuellement sous l'alias `plt`) est l'outil standard pour la représentation graphique.

– La construction d'un graphique suit généralement une séquence ordonnée : création de la figure, tracé des données, ajout de métadonnées (titres, légendes, grille) et enfin affichage.

```
import matplotlib.pyplot as plt

plt.figure() # Creation d'une nouvelle fenetre
# Trace de la vitesse en fonction du temps
plt.plot(temps, vitesse, label="Vitesse (m/s)", color="blue")

plt.title("Evolution de la vitesse")
plt.xlabel("Temps (s)")
plt.ylabel("Vitesse (m/s)")
plt.grid()       # Affiche la grille
plt.legend()     # Affiche la legende definie dans plot()
plt.show()       # Bloque l'execution et affiche la fenetre
```

– D'autres types de tracés sont disponibles pour des besoins spécifiques : `plt.semilogx` et `plt.loglog` pour les échelles logarithmiques (Bode), `plt.scatter` pour les nuages de points, ou `plt.hist` pour les histogrammes.

6. Algorithmes Gloutons et Dichotomiques

6.1. Algorithmes gloutons et optimisation

– Les algorithmes gloutons s'inscrivent dans le cadre de la résolution de **problèmes d'optimisation**. Un problème d'optimisation consiste à choisir, parmi un ensemble de solutions possibles, celle qui maximise un gain (fonction-objectif) ou minimise un coût (fonction-coût).

– L'ensemble des solutions qui respectent les contraintes imposées par le problème est appelé l'ensemble admissible. Une solution qui répond au critère d'optimisation (le meilleur score possible) est une solution globale.

– Quelques exemples classiques de problèmes d'optimisation :
  • Le problème du sac à dos (maximiser la valeur des objets emportés sous contrainte de poids).
  • Le problème du rendu de monnaie (minimiser le nombre de pièces pour atteindre une somme).
  • Le problème du voyageur de commerce (minimiser la distance pour visiter un ensemble de villes).

– Le principe d'un algorithme glouton est de faire, à chaque étape, le choix qui semble le meilleur localement (le choix optimal à l'instant t), sans jamais revenir sur une décision prise, dans l'espoir que cette suite de choix locaux mène à l'optimum global. Notez que cela ne garantit pas toujours de trouver la solution optimale absolue, mais fournit souvent une solution approchée acceptable rapidement.

6.2. Algorithmes dichotomiques

– La méthode dichotomique (du grec "couper en deux") est une stratégie de recherche efficace qui consiste à réduire de moitié l'espace de recherche à chaque étape. Elle s'applique principalement dans deux contextes : la recherche dans une liste **triée** et la recherche de racine d'une fonction monotone.

– **Recherche dans une liste triée** : On cherche un élément x_0 dans une liste L triée. On compare x_0 avec l'élément central de la liste. Si x_0 est plus petit, on ne cherche que dans la moitié gauche ; s'il est plus grand, dans la moitié droite. On répète le processus jusqu'à trouver l'élément ou épuiser la liste.

– Cette méthode est beaucoup plus efficace qu'une recherche séquentielle : sa complexité est logarithmique ($O(\log_2 n)$).

```
1  def recherche_dichotomie(L, x0):
2      """ Recherche x0 dans une liste L triee. Renvoie un booleen. """
3      g = 0                # Indice gauche
4      d = len(L) - 1       # Indice droit
5      found = False
6
7      while g <= d and not found:
8          m = (g + d) // 2  # Indice milieu (division entiere)
9          if x0 == L[m]:
10             found = True
11         elif x0 < L[m]:
12             d = m - 1     # On cherche a gauche
13         else:
14             g = m + 1     # On cherche a droite
15
```

```
    return found
```

– **Recherche de racine (Méthode de la bisection)** : Il s'agit de trouver une solution approchée de l'équation $f(x) = 0$ sur un intervalle $[a, b]$. Le principe repose sur le Théorème des Valeurs Intermédiaires (TVI). Si f est continue et que $f(a)$ et $f(b)$ sont de signes opposés ($f(a) \cdot f(b) < 0$), alors il existe au moins une racine dans l'intervalle.

– L'algorithme calcule le milieu $m = \frac{a+b}{2}$. Si $f(a)$ et $f(m)$ sont de signes contraires, la racine est dans $[a, m]$, sinon elle est dans $[m, b]$. On réduit ainsi la taille de l'intervalle par 2 à chaque itération jusqu'à ce que sa largeur soit inférieure à une précision ϵ donnée.

```python
def zero_dichotomie(f, a, b, epsilon):
    """ Trouve une racine de f dans [a, b] avec precision epsilon """
    val_g = a
    val_d = b
    while (val_d - val_g) > epsilon:
        m = (val_g + val_d) / 2
        if f(val_g) * f(m) <= 0:  # Changement de signe a gauche
            val_d = m
        else:                     # Changement de signe a droite
            val_g = m
    return (val_g + val_d) / 2
```

– La bibliothèque `scipy` propose une implémentation optimisée de cet algorithme via la fonction `bisect`.

```python
import scipy.optimize as spo
# Recherche de la racine de f entre 1 et 2
racine = spo.bisect(f, 1, 2)
```

7. Récursivité

7.1. Principe des fonctions récursives

– Une fonction est dite récursive si son corps contient un ou plusieurs appels à elle-même. C'est une méthode de résolution de problèmes consistant à décomposer un problème complexe en sous-problèmes de même nature mais de taille réduite.

– Pour qu'une fonction récursive soit valide et se termine, deux conditions sont impératives :
  • L'existence d'un ou plusieurs **cas de base** (ou conditions d'arrêt) qui sont traités sans appel récursif.
  • Une progression stricte des appels récursifs vers ce cas de base (souvent via un paramètre entier décroissant ou la taille d'une liste qui diminue).

– L'exemple canonique est le calcul de la factorielle $n! = n \times (n-1)!$ avec $0! = 1$.

```
1 def factorielle(n):
2     if n == 0:            # Cas de base
3         return 1
4     else:                 # Appel recursif
5         return n * factorielle(n - 1)
```

7.2. Analyse des fonctions récursives

– **Gestion des appels (Pile d'exécution)** : Lorsqu'une fonction s'appelle elle-même, l'interpréteur suspend l'exécution courante et empile le contexte (variables locales, paramètres, adresse de retour) dans une structure de données appelée *pile d'exécution* (call stack). Lorsque le cas de base est atteint, les résultats sont renvoyés successivement ("dépilés") lors de la remontée. Si la profondeur de récursion est trop importante, on risque une erreur de type `RecursionError` (débordement de pile ou *stack overflow*).

– **Récursivité terminale** : Une fonction est dite récursive terminale si l'appel récursif est la toute dernière instruction exécutée (aucune opération n'est effectuée sur le résultat renvoyé par l'appel). Cette forme est théoriquement optimisable par le compilateur pour ne pas consommer de pile (ce n'est cependant pas le cas en Python standard).

– **Complexité** : La complexité temporelle se calcule souvent en établissant une relation de récurrence sur le nombre d'opérations $C(n)$. Par exemple, pour la factorielle, $C(n) = C(n-1) + O(1)$, ce qui conduit à une complexité linéaire $O(n)$. Pour une dichotomie, la division de la taille du problème par 2 à chaque étape mène à une complexité logarithmique $O(\log n)$.

– **Récursif vs Itératif** : Tout programme récursif peut être transformé en version itérative (avec des boucles).
  • *Avantages du récursif* : Code souvent plus élégant, lisible et proche de la définition mathématique (ex : suites, structures arborescentes).
  • *Inconvénients* : Coût mémoire (pile) et surcoût temporel lié aux appels de fonctions. L'itératif est généralement plus efficace en Python.

7.3. Exemple : Suite de Fibonacci

– La définition mathématique est $F_0 = 0, F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$.

– L'implémentation récursive "naïve" est extrêmement inefficace (complexité exponentielle) car elle recalcule de nombreuses fois les mêmes termes.

```python
def fibo_rec(n):
    if n < 2: return n
    return fibo_rec(n-1) + fibo_rec(n-2)
```

– Une version itérative (ou utilisant la mémoïsation) est indispensable pour calculer des termes de rang élevé.

7.4. Suite de Syracuse

– La suite de Syracuse est définie pour un entier $N > 0$ par : $u_0 = N$, et $u_{n+1} = u_n/2$ si u_n est pair, $3u_n + 1$ sinon. La conjecture affirme que la suite finit toujours par atteindre le cycle trivial (4, 2, 1).

– C'est un excellent exercice pour pratiquer la récursivité terminale en passant les résultats intermédiaires en arguments.

8. Matrices de pixels et images

8.1. Introduction : formats de représentation d'image

- Il existe deux modes de codage numérique. Le mode **vectoriel** décrit les formes par des propriétés mathématiques (zoom infini sans perte, ex : PDF, SVG). Le mode **matriciel** (ou *bitmap*) repose sur une grille de pixels (ex : PNG, JPEG).
- Un pixel (*picture element*) est le plus petit élément constitutif. Une image matricielle se caractérise par sa **définition** (nombre total de pixels) et sa **résolution** (nombre de pixels par unité de longueur, en DPI ou PPP).
- La colorimétrie utilise principalement le modèle **RVB** (Rouge, Vert, Bleu). Chaque couleur est codée sur un octet (de 0 à 255). Un pixel est ainsi un triplet (R, V, B). Le noir correspond à $(0, 0, 0)$ et le blanc à $(255, 255, 255)$.

8.2. Manipulation des matrices de pixels

- En Python, une image est traitée comme un tableau Numpy de dimension 3 (hauteur, largeur, 3 composantes). On utilise `matplotlib.image` (alias `img`) pour la lecture et `pyplot` pour l'affichage.

```
1 import matplotlib.image as img
2 import matplotlib.pyplot as plt
3
4 im = img.imread("image.png") # Charge l'image en ndarray
5 plt.imshow(im)               # Prepare l'affichage
6 plt.show()                   # Affiche la fenetre
```

- On accède à un pixel via `im[i, j]` et on le modifie en affectant un nouveau triplet : `im[i, j] = [100, 50, 12]`.

8.3. Transformation d'images

- **Symétrie et Rotation** : Ces transformations consistent à réorganiser les indices des pixels. Par exemple, une symétrie horizontale inverse l'ordre des colonnes.
- **Niveau de gris** : On transforme une image RVB en une matrice de dimension 2. La valeur de chaque pixel est une moyenne pondérée des trois composantes. Selon la norme 709, on utilise : $G = 0.2125 \cdot R + 0.7154 \cdot V + 0.0721 \cdot B$.
- Pour afficher une image en niveaux de gris, il faut spécifier la palette de couleurs dans `imshow` : `plt.imshow(im_gris, cmap='gray')`.

8.4. Modification par convolution : filtrage

- Le filtrage consiste à modifier la valeur d'un pixel en fonction de ses voisins. On utilise une petite matrice appelée **noyau de filtrage** (ou filtre).
- Le nouveau pixel est obtenu par le produit de convolution : c'est la somme des pixels voisins pondérée par les coefficients du filtre. Pour conserver la luminosité, on divise souvent le résultat par la somme des coefficients du filtre.
- Exemples de filtres classiques :
  - `Flou` : Matrice remplie de 1 (moyenneur). Plus la taille du filtre ($n \times n$) est grande, plus le flou est prononcé.

- Rehausseur de contraste : Noyau avec des valeurs négatives en périphérie et une valeur forte au centre.

– **Gestion des bords** : Pour un pixel en bordure, certains voisins manquent. La solution la plus simple consiste à ignorer les bords de l'image (balayage de la ligne 1 à $h-1$).

– **Masques** : On peut appliquer un filtre sur une zone spécifique en utilisant une matrice masque (image binaire). Le résultat final est une combinaison : $I_{finale} = M \cdot I_{floue} + (1 - M) \cdot I_{initiale}$.

8.5. Détection de contours

– La détection de contour repose sur l'identification des changements brutaux de couleur ou de contraste entre pixels voisins.

– On calcule pour chaque pixel une "distance" (gradient) par rapport à ses voisins directs (p_1, p_2, p_3, p_4). Une formule courante est la distance euclidienne des différences : $d = \sqrt{(p_1 - p_3)^2 + (p_2 - p$

– On compare cette distance à une **valeur seuil** : si $d > seuil$, le pixel appartient à un contour et est tracé en blanc ; sinon, il est laissé en noir.

Document en cours d'édition jusqu'en avril 2027.