Chapitre :
Programmation

I. Récursivité

La récursivité est un principe assez général, qui ne se limite pas à la programmation et l'algorithmique. Ce principe décrit tout objet ou concept qui fait référence à lui même pour se définir.
En informatique, une fonction ou plus généralement un algorithme qui contient un ou des appel(s) à lui-même est dit récursif.

1. Le problème de la somme des n premiers entiers

La somme des n premiers entiers est la somme :

$$0 + 1 + 2 + \cdots + n \tag{1}$$

Une solution pour calculer cette somme consiste à utiliser une boucle `for` pour parcourir tous les entiers `i` entre `0` et `n`, en s'aidant d'une variable locale `s` pour accumuler la somme des entiers de `0` à `i`. On obtient par exemple le programme Python suivant :

```
def somme(n):
  s = 0
  for i in range(n+1):
    s += i
  return s
```

Une autre manière d'aborder ce problème est de définir une fonction mathématique $somme(n)$ de la manière suivante :

$$somme(n) = \begin{cases} 0 & \text{si } n = 0 \\ n + somme(n-1) & \text{si } n > 0 \end{cases}$$

En effet, pour $n = 0$, $somme(0)$ est égale à 0 et pour n entier strictement positif, $somme(n)$ est égale à $n + somme(n-1)$. Il s'agit d'une définition **récursive**, la définition de $somme(n)$ fait appel à $somme(n-1)$.
Cette définition est directement programmable en Python, comme le montre le code ci-dessous :

```
def somme(n):
  if n == 0:
    return 0
  else:
    return n + somme(n-1)
```

La fonction ainsi obtenue est une **fonction récursive**, l'appel à `somme(n-1)` dans le corps de la fonction est un **appel récursif**.
Par exemple, l'évaluation de l'appel à `somme(3)` peut se représenter à l'aide de l'**arbre d'appels** suivant :

```
somme(3) = return 3 + somme(2)
                        |
                    return 2 + somme(1)
                                |
                            return 1 + somme(0)
                                        |
                                    return 0
```

Pour calculer la valeur renvoyée par `somme(3)`, il faut d'abord appeler `somme(2)` qui fait appel à `somme(1)` qui à son tour nécessite un appel à `somme(0)`.
Le dernier appel se termine directement en renvoyant la valeur 0. Le calcul de `somme(3)` se fait « à rebours ». L'arbre d'appels devient :

```
somme(3) = return 3 + somme(2)
                        |
                    return 2 + somme(1)
                                |
                            return 1 + 0
```

L'appel à `somme(1)` peut alors se terminer et retourner 1. Ainsi on a :

```
somme(3) = return 3 + somme(2)
                        |
                    return 2 + 1
```

Enfin l'appel à `somme(2)` peut retourner la valeur 3, ce qui permet à `somme(3)` de se terminer en retournant le résultat `3+3`.

```
somme(3) = return 3 + 3
```

On obtient bien au final la valeur `6` attendue.

2. Formulations récursives

a. Cas de base et cas récursifs

Une formulation d'une fonction récursive est toujours constituée de un ou plusieurs **cas de base (ou conditions d'arrêt)** et de un ou plusieurs **cas récursifs**.
Les cas récursifs sont ceux qui retournent à la fonction en train d'être définie et les cas de base sont ceux qui donnent directement un résultat.
Dans notre exemple, il y a un cas de base :

$$somme(0) = 0$$

et un cas récursif :

$$somme(n) = somme(n-1) + n$$

b. Cas multiples

Il est également possible de définir une fonction avec plusieurs cas récursifs.
Prenons comme exemple la fonction $puissance(x,n)$ qui calcule x^n :

$$x^n = \underbrace{x \times \cdots \times x}_{n \text{ facteurs}}$$

avec pour convention que $x^0 = 1$.
En remarquant que $x^n = x \times x^{n-1}$, on obtient la formulation récursive suivante :

$$puissance(x,n) = \begin{cases} 1 & \text{si } n = 0 \\ x \times puissance(x,n-1) & \text{si } n > 0 \end{cases}$$

On peut éviter la multiplication (inutile) $x \times 1$ de la définition précédente en ajoutant un cas de base : $puissance(x,1) = x$. On obtient ainsi la définition suivante avec deux cas de base :

$$puissance(x,n) = \begin{cases} 1 & \text{si } n = 0 \\ x & \text{si } n = 1 \\ x \times puissance(x,n-1) & \text{si } n > 0 \end{cases}$$

Il est également possible de définir la fonction $puissance(x,n)$ avec plusieurs cas récursifs. En effet si n est pair, $x^n = \left(x^{\frac{n}{2}}\right)^2$ et si n est impair $x^n = x \times \left(x^{\frac{n-1}{2}}\right)^2$.
En supposant que l'on dispose de la fonction $carre(x) = x \times x$, on obtient la définition récursive suivante :

$$puissance(x,n) = \begin{cases} 1 & \text{si } n = 0 \\ x & \text{si } n = 1 \\ carre(puissance(x,n/2)) & \text{si } n \text{ est pair} \\ x \times carre(puissance(x,(n-1)/2) & \text{si } n \text{ est impair} \end{cases}$$

L'intérêt de cette dernière définition est de réduire de manière significative le nombre d'appels récursifs.

c. Récursion multiple

Dans l'expression d'une fonction récursive, un même cas récursif peut faire plusieurs appels récursifs. Prenons pour exemple la suite de Fibonacci qui doit son nom à Leonardo Fibonacci. Dans un problème récréatif posé dans l'ouvrage *Liber abaci* publié en 1202, il y décrit la croissance d'une population de lapins d'une manière qui peut se traduire sous la forme suivante :

$$fibonacci(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fibonacci(n-2) + fibonacci(n-1) & \text{si } n > 1 \end{cases}$$

Voici par exemple les premières valeurs de cette fonction :

$$\begin{aligned} fibonacci(0) &= 0 \\ fibonacci(1) &= 1 \\ fibonacci(2) &= fibonacci(0) + fibonacci(1) && = 0 + 1 = 1 \\ fibonacci(3) &= fibonacci(1) + fibonacci(2) && = 1 + 1 = 2 \\ fibonacci(4) &= fibonacci(2) + fibonacci(3) && = 1 + 2 = 3 \\ fibonacci(5) &= fibonacci(3) + fibonacci(4) && = 2 + 3 = 5 \end{aligned}$$

d. Récursion mutuelle

Il est également possible de définir plusieurs fonctions récursives en *même temps*, quand ces fonctions font référence les unes aux autres. On parle alors de définitions **récursives mutuelles**.
Prenons par exemple les deux fonctions ci-dessous permettant de tester si un nombre est pair ou impair.

$$pair(n) = \begin{cases} vrai & \text{si } n = 0 \\ impair(n-1) & \text{si } n > 0 \end{cases}$$

$$impair(n) = \begin{cases} faux & \text{si } n = 0 \\ pair(n-1) & \text{si } n > 0 \end{cases}$$

Remarquons tout de même que cette définition est nettement moins efficace que celle utilisant le reste de la division par 2, elle a seulement pour but de fournir un exemple simple de récursion mutuelle.

e. Définition récursive bien formulée

Il y a quelques règles à respecter lorsque l'on écrit une fonction récursive :

- la récursion doit s'arrêter, c'est-à-dire que l'on finit toujours par arriver à un cas de base ;
- les valeurs utilisées pour appeler la fonction sont toujours dans le domaine de la fonction ;
- il y a bien une définition pour toutes les valeurs du domaine.

Pour la fonction $f(n)$ ci-dessous, la condition d'arrêt n'est jamais vérifiée :

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n + f(n+1) & \text{si } n > 0 \end{cases}$$

Le problème vient du fait que l'on fait appel à la fonction avec un entier supérieur ; pour aboutir à un cas de base, il faut généralement s'approcher d'un cas de base à chaque étape, et non s'en éloigner systématiquement.
La fonction $g(n)$ suivante n'est pas définie pour tous les entiers :

$$g(n) = \begin{cases} 1 & \text{si } n = 0 \\ n + g(n-2) & \text{si } n > 0 \end{cases}$$

En effet, $g(1)$ par exemple ne peut être calculé. Il faudrait donc ajouter un cas de base.

3. Programmer avec des fonctions récursives

a. type de données

La fonction `somme(n)` ne se comporte pas exactement comme la fonction mathématique $somme(n)$.

$$somme(n) = \begin{cases} 0 & \text{si } n = 0 \\ n + somme(n-1) & \text{si } n > 0 \end{cases}$$

```
def somme(n):
  if n == 0:
    return 0
  else:
    return n + somme(n-1)
```

La fonction mathématiques est uniquement définie pour les entiers naturels alors que la fonction `somme(n)` peut être appelée avec une valeur quelconque.
Une première solution consiste à modifier le premier test :

```
def somme(n):
  if n <= 0:
    return 0
  else:
    return n + somme(n-1)
```

Cette solution assure la terminaison de la fonction mais modifie sa spécification.

Une autre solution est de restreindre les appels à la fonction `somme(n)`.

```
def somme(n):
  assert n >= 0
  if n == 0:
    return 0
  else:
    return n + somme(n-1)
```

Cette solution est correcte mais a le défaut de faire le test `n >= 0` à chaque appel de `somme(n)` alors que `n` sera positif une fois le premier test réalisé.
Une solution pour éviter ces tests inutiles est de définir une deuxième fonction :

```
def somme(n):
  assert n >= 0
  return somme_bis(n)

def somme_bis(n):
  if n == 0:
    return 0
  else:
    return n + somme_bis(n-1)
```

b. Modèle d'exécution

Il faut savoir que généralement les langages de programmation limitent le nombre d'appels récursifs, entre autres pour des raisons de limitation d'utilisation de mémoire : tant que les valeurs ne sont pas retournées, on augmente une pile (ou liste) de calculs en attente en mémoire (voir l'arbre d'exécution vu tout au début). Il est possible généralement d'augmenter le nombre limite d'appels récursifs, mais cela peut s'avérer dérisoire pour des fonctions qui font des appels récursifs massifs.
Il vaut donc mieux chercher à réduire drastiquement les appels récursifs. Voir par exemple le cas de la fonction puissance, dont la dernière définition est plus efficace que la première, naïve.

II. Diviser pour régner

La stratégie diviser pour régner, *divide and conquer* en anglais, désigne la façon de procéder d'algorithmes qui possèdent une structure récursive de telle sorte qu'ils s'appliquent à des sous-problèmes de taille moindre puis, une fois les solutions de ceux-ci obtenues, celles-ci sont combinées pour aboutir à la solution du problème global.
Autrement dit, un algorithme diviser pour régner procède en 3 étapes :

Diviser : Découper un problème initial en sous-problèmes strictement inférieurs ;

Régner : Résoudre les sous-problèmes récursivement (ou directement s'ils sont assez petits) ;

Combiner : Calculer une solution au problème initial à partir des solutions des sous-problèmes.

L'exemple le plus courant d'algorithme utilisant totalement ce principe est celui du **tri fusion** que nous verrons en exercice, avec d'autres.
En première, nous avons vu un algorithme, celui de **recherche par dichotomie**, qui applique une méthode de division. Il n'est pourtant pas vraiment dans la classe des algorithmes diviser pour régner (même si certains l'y mettent), car il ne fait que la division : il résout un (seul) problème plus simple à chaque itération ; il ne règne donc pas, et il n'y a rien à combiner.
Un autre tri, le **tri rapide** (*quick sort*) (qui n'est pas au programme de NSI), divise et règne, mais ne combine pas.
Pour résumer les idées dans un tableau :

Algorithme	Diviser	Régner	Combiner
Dichotomie	Pour chercher x dans le tableau trié $T[1,..n]$: si $x < T[n/2]$ on cherche dans $T[1,..n/2]$ sinon on cherche dans $T[n/2+1,..n]$	-	-
Tri fusion	on découpe le tableau $T[1,..n]$ à trier en deux sous-tableaux $T[1,..n/2]$ et $T[n/2+1,..n]$	on trie les deux sous-tableaux $T[1,..n/2]$ et $T[n/2+1,..n]$	on fusionne les deux sous-tableaux triés
Tri rapide	on choisit un élément du tableau au hasard qui sera « pivot » et on permute tous les éléments de manière à placer à gauche du pivot les éléments qui lui sont inférieurs, et à droite ceux qui lui sont supérieurs	on trie les deux moitiés de part et d'autre du pivot	-

À savoir : la complexité de l'algorithme de tri fusion est $O(n\log_2(n))$. En effet :
Pour l'algorithme de fusion seule, on a deux tableaux en entrée. Soit m la somme de leurs tailles. Pour les fusionner, on fait m comparaisons (à chaque fois, un des indices augmente).
La complexité de l'algorithme de fusion est donc $O(m)$.
À chaque « étape » (ou niveau d'appel récursif) la somme totale des tailles de tous les tableaux à fusionner deux par deux est exactement n. Donc à chaque étape la complexité de l'ensemble des appels à l'algorithme de fusion est $O(n)$.
Le nombre d'étapes est égal aux nombres de fois qu'il faut diviser n pour arriver à 1, c'est à dire $\log_2(n)$.
La complexité de l'algorithme de tri fusion est donc $O(n\log_2(n))$.

On rappelle que le coût des tris par sélection et par insertion ont une complexité d'ordre $O(n^2)$.

III. Modules

Lorsque l'on apprend à programmer, les codes sources sont relativement simples et on écrit ces derniers très souvent de manière linéaire. Si l'on veut réutiliser du code déjà existant, on utilise la méthode magique du «copier-coller». Sur des projets avec un nombre de lignes beaucoup plus conséquent, cela pose pas mal de problèmes, notamment la difficulté de maintenance.
Le principe de *modularité* consiste à un découpage des différents aspects du programme et des différentes tâches qui doivent être accomplies de manière que chacune puisse être réalisée indépendamment des autres.

1. Utiliser une bibliothèque

Dans Python, une *bibliothèque* (ou *librairie*) est un ensemble de *modules* permettant d'ajouter des possibilités étendues dans une thématique donnée. C'est une sorte de boîte à outils qui apporte ce qui n'existe pas de manière basique dans le langage.
Un module est un simple fichier Python qui contient des collections de fonctions et de variables globales et qui a une extension `.py`.
On peut citer quelques bibliothèques communes :

- **PIL** pour la manipulation et le traitement d'images.
- **Pygame** pour la création de jeu 2D.
- **Django** pour le développement WEB.
- **Tkinter** pour l'affichage graphique.

On aurait pu en citer de nombreuses autres comme, **Math**, **Random**, **Time**, ...
Nous allons prendre des exemples dans le domaine du traitement d'image et nous intéresser à la bibliothèque PIL. La première chose à faire est de se renseigner sur les possibilités proposées par cette bibliothèque et pour cela, il faut **lire la documentation** de cette dernière. Ce document est souvent en anglais et possède un nombre important de pages, cependant c'est la seule solution pour pouvoir utiliser correctement la bibliothèque.
Dans notre cas, la documentation se trouve sur le site internet suivant :

`https://pillow.readthedocs.io/en/stable/`

Nous allons illustrer quelques exemples de cette bibliothèque à l'aide de l'image `tiger.jpg` que l'on souhaite transformer en niveau de gris et enregistrer en format PNG :

Essayons de décomposer nos contraintes pour chercher dans la documentation les informations souhaitées :

- Comment charger une image ?
- Existe-t-il une fonctionnalité pour convertir une image en nuance de gris ?
- Comment sauvegarder une image et choisir son format ?

Après lecture de la documentation, nous avons nos réponses :

- Au début du tutoriel, dans la section « *Using the Image class* », on nous informe que la lecture d'une image se fait à l'aide de la fonction `open()` du module `Image`.

```
from PIL import Image
img_depart = Image.open('tiger.jpg')
```

- Dans la section suivante, « *Reading and writing images* », on nous apprend que la classe `Image` a une méthode `save()`. De plus, lors de la sauvegarde des fichiers, le nom devient important. A moins que vous ne spécifiez le format, la bibliothèque utilise l'extension du nom de fichier pour découvrir quel format de stockage de fichier utiliser.

```
img_arrivee.save('tiger_ndg.png')
```

- Dans la section « *Color transforms* », la méthode `convert()` est présentée. Elle permet de convertir des images dans différentes représentations. Son utilisation est détaillée dans le chapitre « *Image Module* » ; la nuance de gris est obtenue à l'aide du mode `L`.

```
img_arrivee = img_depart.convert('L')
```

```
from PIL import Image

img_depart = Image.open('tiger.jpg')

# on convertit notre image de départ en niveau de gris
img_arrivee = img_depart.convert('L')

# on enregistre notre nouvelle image
img_arrivee.save('tiger_ndg.png')

# on libère la mémoire
img_depart.close()
img_arrivee.close()
```

2. Utiliser une API (web)

Une API, de l'anglais « *Application programming interface* », est une *interface de programmation d'application*, c'est-à-dire un moyen mis en place par une application pour que d'autres applications puissent interagir simplement avec. Nous utilisons alors les fonctions et les méthodes publiques.
Tout comme les bibliothèques, il existe de nombreuses API, parmi elles, les API Web sont de plus en plus nombreuses.
À titre d'exemple, montrons une utilisation de l'API du site de jeu d'échec https://lichess.org/.
La documentation concernant cette API se trouve à la page :

https://lichess.org/api

Pour pouvoir utiliser toutes ses fonctionnalités il est nécessaire d'obtenir au préalable une clef d'authentification (pour le protocole OAuth), mais il y a quelques requêtes que l'on peut faire sans.
Nous souhaitons obtenir la position du problème du jour proposée sur le site. On trouve l'URL à utiliser dans la page de documentation ; il s'agit de : https://lichess.org/api/puzzle/daily.

La documentation nous informe que les données retournées par le site sont au format JSON.

Pour effectuer des requêtes en Python, le plus simple est d'utiliser un module comme `requests`.
La fonction `get` fait la requête et retourne la réponse obtenue.
Sur cette réponse, la méthode `json()` produit un dictionnaire (dont les valeurs peuvent aussi être des dictionnaires). Ainsi :

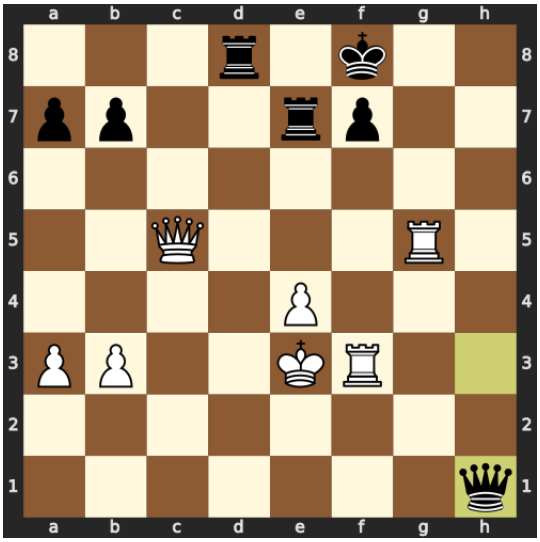
```
>>> import requests
>>> r = requests.get("https://lichess.org/api/puzzle/daily")
>>> data = r.json()
>>> data
{'game': {'id': 'OvCv8BDk', 'perf': {'key': 'bullet', 'name': 'Bullet'},
 ↪  'rated': True, 'players': [{'userId': 'revebe', 'name': 'revebe
 ↪  (1647)', 'color': 'white'}, {'userId': 'academik116', 'name':
 ↪  'academik116 (1544)', 'color': 'black'}], 'pgn': 'e3 e5 c4 Nf6 h3 d6
 ↪  a3 g6 b3 Bg7 Bb2 O-O g4 c6 Bg2 Nbd7 Nc3 Qe7 Nge2 e4 Ng3 d5 cxd5 cxd5
 ↪  Qc2 Ne5 O-O Nf3+ Bxf3 exf3 Nb5 h5 g5 Bxh3 gxf6 Bxf6 Bxf6 Qxf6 Rfe1 h4
 ↪  Nf1 Qg5+ Ng3 hxg3 e4 gxf2+ Kxf2 Qg2+ Ke3 dxe4 Nd6 Rad8 Nxe4 Rfe8 Qc4
 ↪  Bf5 d3 Qg4 Rg1 Qh3 Raf1 Bxe4 dxe4 g5 Rxg5+ Kf8 Qc5+ Re7 Rxf3 Qh1',
 ↪  'clock': '2+1'}, 'puzzle': {'id': 'PZH1h', 'rating': 1791, 'plays':
 ↪  66368, 'initialPly': 69, 'solution': ['f3f7', 'f8f7', 'c5f5', 'f7e8',
 ↪  'g5g8'], 'themes': ['endgame', 'attraction', 'long', 'mateIn3',
 ↪  'sacrifice', 'pin']}}
>>> pgn = data["game"]["pgn"]
>>> pgn
'e3 e5 c4 Nf6 h3 d6 a3 g6 b3 Bg7 Bb2 O-O g4 c6 Bg2 Nbd7 Nc3 Qe7 Nge2 e4
 ↪  Ng3 d5 cxd5 cxd5 Qc2 Ne5 O-O Nf3+ Bxf3 exf3 Nb5 h5 g5 Bxh3 gxf6 Bxf6
 ↪  Bxf6 Qxf6 Rfe1 h4 Nf1 Qg5+ Ng3 hxg3 e4 gxf2+ Kxf2 Qg2+ Ke3 dxe4 Nd6
 ↪  Rad8 Nxe4 Rfe8 Qc4 Bf5 d3 Qg4 Rg1 Qh3 Raf1 Bxe4 dxe4 g5 Rxg5+ Kf8
 ↪  Qc5+ Re7 Rxf3 Qh1'
```

On voit que les données (`data`) contiennent plusieurs informations, notamment la partie d'où est tirée le problème, y compris le pgn, autrement dit le déroulement de la partie jusqu'à la position du problème, mais également des données sur le problème (son identité, son niveau, etc. et même la solution).
En installant la librairie python-chess, on peut ajouter quelques lignes de code pour obtenir l'image de cette position de problème :

```python
import requests
import chess
import chess.svg

r = requests.get("https://lichess.org/api/puzzle/daily")
data = r.json()
pgn = data["game"]["pgn"] # récupération du pgn

board = chess.Board() # Création d'un plateau en position initiale
coups = pgn.split(" ") # On transforme le pgn en liste de coups
                       # en séparant par les espaces
for coup in coups:
    board.push_xboard(coup) # On joue les coups sur le plateau
```

```
lastmove = board.pop() # On enregistre le dernier coup (en l'annulant)
board.push(lastmove) # on rejoue le dernier coup

# Par suite, on utilise le module svg de la librairie
# pour obtenir une image :
svgfile = chess.svg.board(
    board,
    orientation = board.turn,
    colors={"square light": "#fff8dc",
            "square dark": "#8b5a2b",
            },
    lastmove=lastmove,
    size=500,
    )

with open("position.svg", "w") as f:
    f.write(svgfile) # On écrit le contenu dans un fichier
```

L'image obtenue est la suivante (ici transformée en fichier .png) :



Il est à noter que, souvent, l'utilisation des API se fait avec une adresse à laquelle on ajoute des arguments supplémentaires pour la requête. Par exemple, pour connaître le top 3 des joueurs de blitz du site, la requête est la suivante :

```
>>> requests.get("https://lichess.org/api/player/top/3/blitz").json()
{'users': [{'id': 'fairchess_on_youtube', 'username':
↪  'FairChess_on_YouTube', 'perfs': {'blitz': {'rating': 3040,
↪  'progress': -22}}, 'title': 'GM'}, {'id': 'iliterallydontcare97',
↪  'username': 'ILiterallyDontCare97', 'perfs': {'blitz': {'rating':
↪  2994, 'progress': -16}}}, {'id': 'athena-pallada', 'username':
↪  'athena-pallada', 'perfs': {'blitz': {'rating': 2993, 'progress':
↪  -3}}, 'title': 'GM', 'patron': True}]}
```

Selon les API, les arguments peuvent être donnés sous forme de variables de la méthode GET. Par exemple, sur le site https://data.education.gouv.fr/, pour obtenir l'adresse et la géolocalisation des lycées généraux de Strasbourg, l'URL est la suivante :

```
https://data.education.gouv.fr/api/records/1.0/search/?dataset=fr-en-adresse-et-g
↪ eolocalisation-etablissements-premier-et-second-degre&refine.libelle_commune=
↪ Strasbourg&refine.nature_uai_libe=LYCEE+D+ENSEIGNEMENT+GENERAL
```

3. Créer un module

Lors de l'implémentation de structure particulières (pile, file, arbre, graphe, ...), nous allons implémenter nos propres modules. Il ne faudra surtout pas négliger la saisie de commentaire pour bien documenter son code.

IV. Programmation orientée objet

L'année dernière, vous avez appris à représenter des données à l'aide de différents types construits : p-uplets (tuples), p-uplets nommés, tableaux et dictionnaires.
Le paradigme de *programmation objet* que nous allons présenter dans ce chapitre, fournit une notion de *classe* qui permet de définir des structures de données composites et de structurer le code d'un programme.

1. Classes

Une **classe** définie et nomme une structure de données qui vient s'ajouter aux structures de base du langage. La structure définie par une classe peut regrouper plusieurs composantes de natures variées appelées *attributs* et chacune dotée d'un nom.

a. Description d'une classe

Supposons que l'on souhaite manipuler des triplets d'entier représentant le temps en heures, minutes, secondes. On peut définir une structure `Chrono` contenant trois attributs appelés `heures`, `minutes` et `secondes`. :

Chrono
heures
minutes
secondes

Python permet la définition de cette structure `Chrono` sous la forme d'une classe avec le code suivant :

```
class Chrono:                    <---- mot clef class suivi du nom et :
    '''une classe pour représenter un temps mesuré
    en heures, minutes et secondes'''
    def __init__(self, h, m, s):  <---- fonction __init__
        self.heures = h
        self.minutes = m          <-- affectation des valeurs aux attributs
        self.secondes = s
```

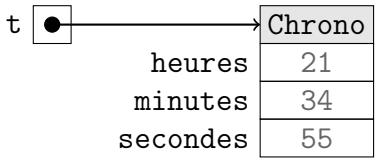
La définition d'une nouvelle classe est introduite par le mot clef `class`, suivi du nom de la classe et du symbole : . Le nom de la classe commence par une lettre majuscule.
La fonction `__init__`, dont nous détaillerons la construction par la suite, possède un premier paramètre appelé `self` ainsi que trois instructions de la forme `self.a = ...` qui permettent d'affecter à chaque attribut sa valeur.

b. Création d'un objet

Une fois la classe `Chrono` définie, on peut créer un élément correspondant à cette structure appelé *objet* ou *instance* de la classe `Chrono`.
On peut par exemple définir et affecter à la variable `t` un objet représentant le temps « 21 heures, 34 minutes et 55 secondes » de la manière suivante :

```
>>> t = Chrono(21,34,55)
```

Tout s'écrit comme si le nom de la classe était une fonction.
La variable `t` contient un pointeur vers le bloc mémoire qui a été alloué à cet objet. On obtient la situation suivante :



c. Manipulation des attributs

On peut accéder aux attributs d'un objet `t` avec la notation `t.a` où `a` désigne le nom de l'attribut visé. Les attributs en Python sont mutables.

```
>>> t.heures
21
>>> t.heures += 1
>>> t.heures
22
```

Une classe peut également définir des *attributs de classe* , dont la valeur est attachée à la classe elle-même.
On peut consulter de tels attributs depuis n'importe quelle instance, ou depuis la classe elle-même.

```
class Chrono:
  heure_max = 24
  ...
```

```
>>> t = Chrono(21, 34, 55)
>>> (t.heure_max, Chrono.heure_max)
(24, 24)
```

2. Méthodes : manipuler les données

Dans le paradigme de la programmation objet, un programme manipulant un objet n'est pas censé accéder à la totalité de son contenu mais uniquement a une interface constitué de fonctions dédiées appelées les *méthodes* de cette classe.

a. Utilisation d'une méthode

L'appel à une méthode `texte` s'appliquant à l'objet `t` qui affiche une chaîne de caractère décrivant le temps représenté est réalisé ainsi :

```
>>> t.texte()
21h 34m 55s
```

Cette notation utilise la même notation pointé que l'accès aux attributs de `t`, mais fait apparaître une paire de parenthèses comme pour l'appel d'une fonction sans paramètre.
L'appel à une méthode `avance` faisant avancer `t` d'un certains nombre de secondes passé en paramètre s'écrit de la manière suivante :

```
>>> t.avance(5)
>>> t.texte()
21h 35m 00s
```

Comme pour les fonctions, les paramètres autre que l'objet principal `t` apparaissent entre parenthèses et séparés par des virgules.

b. Définition d'une méthode

Une méthode d'une classe peut être vu comme une fonction ordinaire à ceci près qu'elle doit nécessairement avoir pour premier paramètre un objet de cette classe. Par convention, ce premier paramètre est systématiquement appelé `self`. Les méthodes `texte` et `avance` peuvent être définies de la manière suivante :

```
def texte(self):
  return (str(self.heures)+'h'
      +str(self.minutes)+'m'
      +str(self.secondes)+'s')

def avance(self,s):
  self.secondes += s
  # depassement secondes
  self.minutes += self.secondes//60
  self.secondes = self.secondes%60
  #depassement minutes
  self.heures += self.minutes//60
  self.minutes = self.minutes%60
```

c. Méthodes particulières

La construction d'un nouvel objet avec une expression comme `Chrono(21, 34, 55)` déclenche deux choses :

1. la création de l'objet lui-même
2. l'appel à une méthode spéciale, appelée *constructeur*, chargé d'initialiser les valeurs des attributs. En Python, il s'agit de la méthode `__init__`.

La particularité de cette méthode est la manière dont elle est appelée, directement par l'interprète Python en réponse à une opération particulière.

Autres méthodes particulières en Python

Il existe en Python un certain nombre d'autres méthodes particulières, chacune avec un nom fixé et entouré de deux paires de `_`. Elles permettent d'alléger ou d'uniformiser la syntaxe. Le tableau suivant en donne quelques exemples.

méthode	appel	effet
`__str__(self)`	`str(t)` ou `print(t)`	retourne une chaîne de caractère décrivant `t`
`__lt__(self,u)`	`t < u`	retourne `True` si `t` est strictement plus petit que `u`
`__len__(self)`	`len(t)`	retourne un nombre entier définissant la taille de `t`
`__contains__`	`x in t`	retourne `True` si `x` est dans la collection `t`
`__getitem__(self,i)`	`t[i]`	retourne le `i`-ième élément de `t`

On peut alors ajouter la définition suivante à la classe `Chrono`

```
  def __str__(self):
    return self.texte()
```

3. Encapsulation

Dans la philosophie objet, l'interaction avec les objets d'une classe se fait avec les méthodes de l'interface. L'utilisateur extérieur n'est pas censé accéder aux attributs ou aux autres méthodes.

Dans certains langage de programmation, comme le C++ ou le Java, on peut empêcher cet accès. En Python, on précède les attributs ou les méthodes qui ne font pas parties de l'interface par `_` mais rien n'empêche que l'utilisateur extérieur y accèdent.

Dans notre classe `Chrono`, il vaut donc mieux appeler nos trois attributs `_heures`, `_minutes` et `_secondes`.

```
class Chrono:
    def __init__(self,h,m,s):
        self._heures = h
        self._minutes = m
        self._secondes = s
```

On peut alors modifier structure de la classe tant que l'interface reste inchangée. En l'occurrence, on pourrait modifier la classe `Chrono` et se contenter d'un attribut `_temps` mesurant le temps en secondes.

```
class Chrono:
    def __init__(self,h,m,s):
        self._temps = 3600*h + 60*m + s
```

Les opérations comme `avance` sont alors simplifiées :

```
  def avance(self,s):
    self._temps += s
```

On pourra alors introduire une méthode qui ne fait pas partie de l'interface mais destinée à être utilisée par les méthodes principales :

```
  def _conversion(self):
    s = self._temps
    return (s//3600, (s//60)%60, s%60)

  def texte(self):
    h,m,s = self._conversion()
    return str(h)+'h '+str(m)+'m '+str(s)+'s'
```

V. Mise au point de programmes

Lorsque l'on exécute un programme, il peut fonctionner comme prévu, mais il peut aussi se passer tout un tas de problèmes différents, de l'arrêt sur une erreur au fait de boucler indéfiniment en passant par des résultats qui ne sont pas ceux qui sont attendus.
La détection des erreurs dans un programme peut être simple, en particulier quand un message d'erreur indique clairement l'endroit qui l'a causé, mais peut aussi dans certains cas être complexe et nécessiter un vrai travail approfondi.

1. Messages d'erreurs

Les langages de programmation possèdent des systèmes de message d'erreur permettant d'identifier les plus simples, liées à une mauvaise utilisation des éléments manipulés.
Les premières erreurs détectées sont les erreurs de syntaxes, recherchées avant même d'exécuter le code. Il s'agit de vérifier que le code est écrit dans les règles syntaxiques du langage. Le système va par exemple détecter certaines erreurs d'indentation, l'oubli de caractères comme les ' :' en fin de ligne d'instruction comme `if`, `for` ou `while`, ou l'utilisation de l'instruction d'affectation `=` au lieu du test d'égalité `==`.
D'autres erreurs sont détectées lors de l'exécution du code. En voici quelques exemples :

- `IndexError: list index out of range` indique une erreur d'indice, par exemple dans une liste, généralement trop grand
- `TypeError:` Utilisation d'un élément d'un mauvais type. On peut trouver plusieurs erreurs de types, comme :
  * `L[1.5]`
    `TypeError: list indices must be integers or slices, not float`
    On a donné un indice flottant, ce qui n'est pas possible dans une liste
  * `L(i)`
    `TypeError: 'list' object is not callable`
    On a mis des parenthèses après le nom d'une liste, comme lorsque l'on appelle une fonction, alors qu'il faut des crochets.
  * `mystere[i]`
    `TypeError: 'function' object is not subscriptable`
    Ici au contraire, on a mis des crochets derrière le nom d'une fonction, comme si on en voulait l'élément d'indice `i`, alors qu'il faut l'appeler en mettant des parenthèses.
  * `a[i]`
    `TypeError: 'int' object is not subscriptable`
    Même chose, avec une variable `a` de type `int` (entier).
  * `i+'a'`
    `TypeError: unsupported operand type(s) for +: 'int' and 'str'`
    On ajoute à tort un entier `i` avec une chaîne de caractère `'a'`
  * `'a'+i`
    `TypeError: can only concatenate str (not "int") to 'str'`
    Même genre de problème mais message différent, car il indique qu'un + après une chaîne de caractère doit nécessairement être suivi d'une chaîne de caractère.
- `print(j)`
  `NameError: name 'j' is not defined`
  La variable `j` n'a pas de valeur à cet endroit du code.

Ce type d'erreur peut se trouver dans la situation suivante, où l'on utilise la valeur d'une variable non définie précédemment dans une fonction :

```
def f():
  t = t+1

f()
```

```
UnboundLocalError: local variable 't' referenced before assignment
```

En effet, `t` n'a pas de valeur dans la fonction `f`, donc t+1 ne peut être calculée, alors qu'elle est locale à la fonction et que l'on est en train de définir la valeur de cette variable.

Il y a évidemment beaucoup d'autres messages d'erreur. Il est important de bien les lire, les identifier et les comprendre pour être capable de corriger le code en conséquence. Un système d'affichage d'erreur n'est jamais parfait ; parfois il n'indique pas réellement la ligne qui pose problème quand par exemple il s'agit d'une parenthèse oubliée un peu plus haut. Mais ces messages sont tout de même d'une grande aide pour résoudre beaucoup de problèmes évidents.

2. Détection d'erreurs plus subtiles

D'autres erreurs ne sont pas détectées par le langage, parce qu'il ne s'agit pas de problèmes de syntaxe, de type ou autre qui sont détectées à l'exécution, mais parce que les calculs effectués ne sont pas ceux qui sont souhaités initialement, ce que ne peut pas deviner la machine.

Une approche classique consiste à **ajouter des traces**, autrement dit afficher des résultats intermédiaires ou des informations lors de l'exécution du code pour en vérifier le fonctionnement. Pour cela, on peut utiliser la fonction `print` pour afficher des valeurs de variables ou d'autres informations, mais aussi l'instruction `assert` pour s'assurer que certaines spécifications sont respectées.

On peut également utiliser un outil de **débogage**, qui permet d'exécuter un programme pas à pas, autrement dit instruction après instruction, avec les valeurs des variables à chaque étape.

3. Plusieurs types de tests

Tester un programme consiste à vérifier qu'il fonctionne dans un maximum de situations possibles. Un **jeu de test**, comme nous en avons vu en première, est un outil utile pour cela, permettant de mettre en défaut les parties testées. Si les tests échouent, il s'agit d'identifier la source du problème et de corriger le code.

- Les **tests unitaires** concernent ceux de petites unités d'un gros programme, comme une fonction.
- Les **tests d'intégration** sont ceux qui permettent de vérifier que des parties différentes d'un programme (un ensemble de fonctions) fonctionnent correctement ensemble.
- Les **tests de performance** permettent de savoir si un programme reste efficace lorsque l'on augmente la taille des données à traiter.
- Les **tests d'utilité** évaluent l'ergonomie du programme, c'est à dire si les utilisateurs parviennent simplement à réaliser les tâches souhaitées avec celui-ci.

Pour les tests unitaires et d'intégration, on distingue plusieurs types de tests :

- Les **tests fonctionnels** qui vérifient la conformité des fonctions à leurs spécifications (types des valeurs en entrée, en sortie, avec leurs contraintes).
  Ces tests ne se soucient pas de l'implémentation, on les appelle parfois tests « boîte noire ».
- Les **tests structurels** qui vérifient le fonctionnement interne du programme. Ces tests sont écrits spécifiquement en fonction de l'implémentation des fonctions, on les appelle parfois tests « boîte blanche ».

Bien entendu, quand on corrige une erreur, d'autres peuvent apparaître, nouvelles ou cachées par les précédentes, il convient donc de recommencer les tests déjà effectués.

Pour créer un **jeu de tests**, il existe plusieurs outils, dont **pytest**. Cet outil permet d'exécuter un

ensemble de fonctions de test contenues dans un fichier. Une fonction de test est une fonction dont le nom commence par `test_` ou termine par `_test`, et que l'on peut placer dans un module dont le nom commence par `Test`. Elle doit contenir une assertion qui est vraie si le test est correct. Dans le cas où la fonction de test doit vérifier qu'une assertion du programme a échoué, on utilise une autre forme de test, la seconde de l'exemple ci-dessous d'un fichier `factorial.py` :

```
import pytest

def fact(n):
    assert n>=0
    return fact_r(n)

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

class TestModule:
  def test_fact(self):
      assert fact(4) == 24 # test OK si l'assertion est vraie

  def test_assert_fact(self):
      with pytest.raises(AssertionError):
          fact(-1)
      # code qui doit faire échouer une assertion
      # d'une fonction.
      # Le test est donc OK si l'assertion est fausse
```

Par suite, la commande `pytest factorial.py` lancée dans un terminal exécute l'ensemble des fonctions de test du fichier `factorial.py` situé dans le répertoire courant et indique celles qui ont réussi et celles qui ont provoqué une erreur.

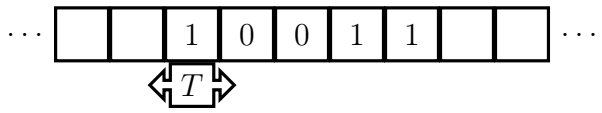
```
============================= test session starts ==============================
platform linux -- Python 3.10.6, pytest-7.1.2, pluggy-1.0.0+repack
rootdir: /media/Fichiers/tmp
collected 2 items

factorial.py ..                                                          [100%]

============================== 2 passed in 0.00s ===============================
```

Il est de bonne pratique de fournir avec chaque module un module de test pour pytest. Ainsi, lorsque l'on modifie le module, on peut rapidement vérifier qu'il passe les tests avec succès : on parle de **non régression**.
On note l'intérêt d'avoir programmé les tests, de ne pas les effectuer à chaque exécution du fichier mais seulement lorsqu'on souhaite tester.

VI. Calculabilité et décidabilité

1. La machine de Turing

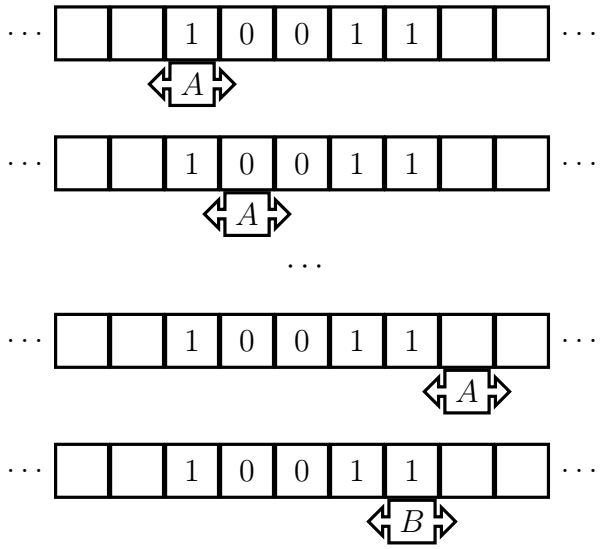
Une **machine de Turing** est un système comportant deux composants : un **ruban** infini et une **tête de lecture/écriture** dont on le munit.



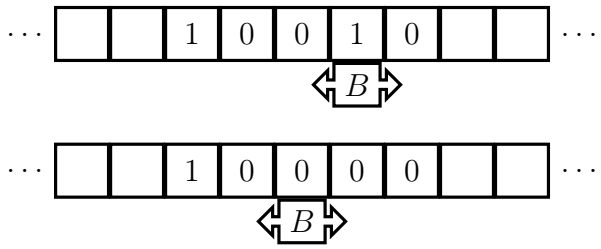
Le ruban est divisé en cases qui chacune peut contenir soit un symbole d'un ensemble fini appelé *alphabet*, soit être vide (blanc). On prendra typiquement comme alphabet les symboles 0 et 1.
La tête peut lire puis écrire (ou effacer) sur la case du ruban qui lui fait face et enfin se déplacer d'une case vers la gauche ou la droite (parfois, on peut considérer que c'est en fait le ruban qui se déplace, ce qui inverse les déplacements). Elle peut également changer d'état, ce qui lui permet de changer de comportement selon celui-ci. Les états possibles sont donnés dans un **registre d'états**, qui est un ensemble fini et contient nécessairement un état initial (de départ) et peut contenir un ou plusieurs états terminaux.
Pour que la machine « fonctionne », il faut ajouter un élément : la **table de transition**. Un programme réalisé par une machine de Turing est en effet décrit par un ensemble de *règles de transition*.
Considérons la table de transition suivante :

état	lu	écrit	déplacement	état suivant
A	0	0	$\rightarrow$	A
A	1	1	$\rightarrow$	A
A	blanc	blanc	$\leftarrow$	B
B	0	1	$\cdot$	F
B	1	0	$\leftarrow$	B
B	blanc	1	$\cdot$	F

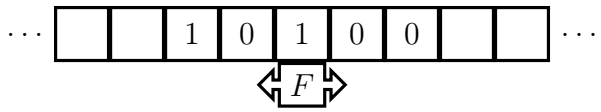
La tête de lecture possède 3 états : A (initial), B et F (pour Fin, c'est un état terminal).
Dans l'état A, la tête va se déplacer tout à droite puis passer à l'état B :



Dans cet état, tant que la machine observe des 1, elle les remplace par des 0 et poursuit son retour vers la gauche :

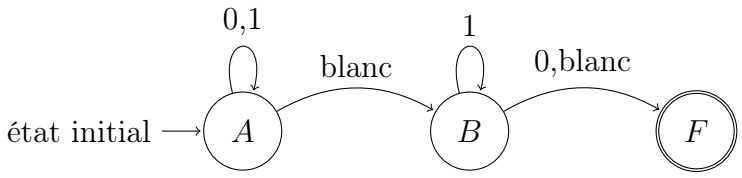


Ce retour en arrière s'arrête dès que la machine lit un 0 ou un blanc, auquel cas ce symbole est remplacé par 1 et la machine passe à son état final F, donc s'arrête.



La valeur initiale sur le ruban était 10011, c'est à dire 19, et le résultat est 10100, c'est à dire 20. La machine de Turing donnée par la table de transition incrémente de 1 l'entier écrit sur le ruban.

Remarque On peut représenter les machines de Turing, ou plutôt leur table de transition, par des graphes orientés et étiquetés, que l'on appelle alors **automates**, montrant les transitions d'état.
La machine de l'exemple peut être représentée par cet automate :



Les étiquettes sur les arcs sont les valeurs lues par la tête de lecture.
Pour qu'il soit complet, il faut rajouter les indications d'opérations d'écriture et de déplacement.

2. Calculabilité

La machine de Turing : tout ne se calcule pas !
Si on appelle *programme* la description d'une suite d'opérations à effectuer mécaniquement, alors un programme peut être représenté par une machine de Turing.
Un nombre est **calculable** (*computable* en anglais, d'après Turing) s'il peut être obtenu par une machine de Turing. Selon la *thèse de Church*, un nombre *calculable* au sens intuitif correspond à un nombre *calculable* par une machine de Turing.
Pour préciser, un nombre est calculable s'il existe une machine de Turing (ou un programme, ce qui est équivalent) qui nous permet d'obtenir un par un la suite des chiffres de son écriture (binaire, ou décimale) qui peut être infinie. Autrement dit, si la machine peut nous donner une approximation à n'importe quelle précision souhaitée.
On peut calculer beaucoup de choses, bien sûr, y compris le nombre π.
On peut alors se poser la question suivante : peut-on tout calculer ? La réponse est non. En fait, de nombreux nombres sont non calculables ; il y en a même bien davantage (en un certain sens, car ces ensembles sont infinis !) que de nombres calculables. Cela sort du cadre du programme de lycée.
De même que pour les nombres, on peut définir ce que sont des **fonctions calculables** : Une fonction f est calculable si, quelle que soit la valeur de u, le nombre $f(u)$ est calculable (avec une machine qui s'arrête). Comme pour les nombres, il y a des fonctions non calculables.

Une idée de la démonstration de cela utilise le fait que le nombres de machines de Turing est dénombrable (on pourrait toutes leur affecter un numéro entier), alors que les fonctions sont non dénombrables (ce qui peut se démontrer à l'aide d'un raisonnement dû à Georg Cantor).
Certains problèmes théoriques peuvent se ramener à la question de la calculabilité d'une fonction donnée.
En voici deux exemples majeurs :

- Est-ce qu'il existe un algorithme (donc un programme, ou une machine de Turing) qui permet de savoir si un énoncé mathématique quelconque est un théorème (que l'on peut démontrer) ou non ?
  La fonction associée prend la valeur 1 pour tout énoncé démontrable et 0 pour tout énoncé non démontrable.
  Cette fonction n'est pas calculable.
- Est-ce qu'il existe un programme qui peut prendre en argument un programme P et qui détermine si l'exécution de P termine ou non ?
  La fonction associée prend la valeur 1 pour tout programme P qui termine, 0 sinon.
  Cette fonction n'est pas calculable.

3. Décidabilité

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 qui permet de répondre par oui ou par non à la question posée par le problème. Sinon on dit que le problème est **indécidable**.
Un exemple de problème de décision, est de déterminer si un nombre entier `n` est pair ou non. Le programme suivant répond à cette question :

```
def pair(n):
    while n>0:
        n = n-2
    return n==0
```

Le problème est donc décidable.
Le **problème de l'arrêt** est le premier problème indécidable exposé par Alan Turing en 1936. Il a démontré qu'il ne pouvait exister un algorithme `arret(f, e)` qui prend en paramètre un algorithme `f` et son entrée `e`, et déterminant si `f` se termine avec l'entrée `e` ou se poursuit indéfiniment.
Supposons qu'une telle fonction Python existe :

```
def arret(f, e):
    """True si la fonction f termine avec l'entrée e et False sinon."""
    pass
```

Nous pouvons alors construire une nouvelle fonction `paradoxe(n)`, qui prend un entier `n` comme paramètre, à partir de `arret` de la manière suivante :

```
def paradoxe(n):
    if arret(paradoxe, n):
        while True:
            pass
    else:
        return n
```

Que ce passe-t-il alors lorsque l'on exécute `paradoxe(0)` ? De deux choses l'une :

- L'exécution se termine et dans ce cas `arret(paradoxe, 0)` va renvoyer `True` ce qui va déclencher une boucle infinie et donc `paradoxe(0)` ne s'arrête pas. On a une contradiction.
- L'exécution ne se termine pas et dans ce cas `arret(paradoxe, 0)` va renvoyer `False` et donc `paradoxe(0)` va renvoyer `0`. On a à nouveau une contradiction.

On en déduit qu'il ne peut exister un algorithme tel que `arret`.
Une vidéo pour terminer (en anglais, mais avec des sous-titres français) :
Proof That Computers Can't Do Everything (The Halting Problem)

VII. Programmation fonctionnelle

1. Divers paradigmes de programmation

Le mot **paradigme** vient du mot grec ancien $\pi\alpha\rho\alpha\delta\epsilon\iota\gamma\mu\alpha$ qui signifie « modèle ». En informatique, ce mot a été adopté pour distinguer les différentes façons d'envisager l'écriture d'un programme.
Il existe de nombreux paradigmes de programmation :

Impératif	Structuré	Fonctionnel
Orienté objet	Concurrent	Distribué
Événementiel	Logique	...

Dans le paradigme **impératif**, un état implicite (la mémoire) est modifié par les instructions. C'est basiquement les langages **machine** ou **assembleur** qui sont sous cette forme. Il utilise entre autres une instruction de la forme « `goto` » qui permet de se déplacer dans les lignes de code (ou les cellules mémoire dans lesquelles est inscrit le code).
Dans le paradigme **structuré**, on s'interdit le « `goto` » et on utilise des conditionnelles plus structurées que dans l'impératif. Ainsi, on peut utiliser les instructions `while`, `for`, etc. ainsi que `break` ou `return`. C'est généralement celui par lequel on commence l'apprentissage de la programmation.
Le paradigme **orienté objet**, avec les concepts de classe et de méthode, ont été présentés précédemment. Ses avantages sont de regrouper données et fonctions, de permettre de restreindre l'accès à des données (attributs ou méthodes privées). Cela permet bien entendu la modularité. On peut également changer la structure de données sans changer l'interface (ce que nous avons déjà pu observer, notamment avec les structures de pile et de file).
Le paradigme **événementiel** est celui qui a été vu avec JavaScript, mais aussi avec le module tkinter de Python. C'est celui qui consiste à déclencher l'exécution d'instructions dès que des événements précis surviennent (un clic quelque part, le déplacement du curseur de la souris, la frappe d'une touche du clavier, etc.).
Le paradigme **concurrentiel** permet de gérer des processus en parallèle. Pour le **distribué**, il s'agit de gérer l'exécution sur plusieurs ordinateurs.
Un paradigme qui en regroupe plusieurs est le paradigme de la **programmation déclarative**. Il s'agit là non pas de définir la manière d'obtenir le résultat mais ce que l'on veut. Ce paradigme regroupe ceux de la **programmation fonctionnelle**, **logique** et par **contrainte**. Les langages utilisant ce genre de paradigme sont dits de haut niveau, par opposition au langage machine qui est le langage de plus bas niveau qui soit. Ils nécessitent cependant un temps d'apprentissage supplémentaire pour bien les maîtriser, car il ne s'agit pas seulement d'un changement de syntaxe, mais bien de penser les programmes.
Nous allons ici développer la **programmation fonctionnelle**. Dans ce paradigme, il n'y a aucune affectation (pourtant très courante en programmation impérative), uniquement des fonctions. Ainsi, contrairement à la programmation impérative, il n'y a pas de gestion d'un état de mémoire, pas de mise à jour des données, pas non plus d'effet de bord (la modification d'une variable globale en est un). Dans ce paradigme, on fait les répétitions à l'aide de la **récursivité**, on n'utilise pas les boucles `while` ou `for`. Il reste cependant l'instruction conditionnelle `if`, et on peut être aidé par l'utilisation des instructions `match` et `case` qui en sont une généralisation. Certaines fonctions de haut niveau (`filter`, `map`, `reduce`, etc.) peuvent être utilisées. D'autre part, les fonctions elles-mêmes peuvent être utilisées comme paramètres des fonctions.

En théorie, tous les langages de programmation sont équivalents, dans le sens où ils permettent les mêmes calculs, d'obtenir ainsi les mêmes choses. Les paradigmes ne sont alors que des manières différentes d'exprimer les instructions à exécuter.
Les langages de programmation n'imposent pas tous un paradigme. Par exemple, le langage Python autorise les programmations impérative, objet, événementielle, concurrentielle mais aussi fonctionnelle, entre autres. Cela permet alors de passer d'un paradigme à l'autre, de faire des mélanges, selon ses habitudes et préférences de programmation, voire selon le type de problème à résoudre.
Nous donnons ci-après quelques exemples d'utilisation de ce paradigme, qui demande un réel temps d'apprentissage pour être compris et maîtrisé.

2. Fonctions en tant que paramètres

a. Fonctions passées en arguments

Les fonctions sont comme des objets en Python, elles peuvent donc être passées en argument à d'autres fonctions.
Dans l'exemple ci-dessous, nous avons créé une fonction `saluer` qui prend une autre fonction comme argument :

```
def crier(texte):
    return texte.upper()

def chuchoter(texte):
    return texte.lower()

def saluer(f):
    return f("Bonjour, je suis créé par une fonction"
             "passée en argument.")
```

```
>>> saluer(crier)
'BONJOUR, JE SUIS CRÉÉ PAR UNE FONCTION PASSÉE EN ARGUMENT.'
>>> saluer(chuchoter)
'bonjour, je suis créé par une fonction passée en argument.'
```

Donnons pour application un exemple avec la fonction de tri `sorted` qui, comme nous l'avons déjà vu, permet de trier les éléments d'une liste par ordre croissant :

```
>>> noms = ['Yang', 'Robert', 'Tom', 'Gates']
>>> sorted(noms)
['Gates', 'Robert', 'Tom', 'Yang']
```

L'ordre croissant correspond ici par défaut à l'ordre alphabétique (pour les chaînes de caractère. On peut cependant classer cette liste selon un autre ordre, par exemple la taille ; pour cela il faut passer la **fonction** correspondante en paramètre, nommé `key` :

```
>>> sorted(noms, key=len)
['Tom', 'Yang', 'Gates', 'Robert']
```

b. Fonctions anonymes

Une *fonction anonyme* est une fonction n'ayant pas de nom. Python propose une notion de fonction anonyme sous la forme d'une construction `lambda` selon la syntaxe :

```
lambda arguments: image
```

La notation `lambda` provient du λ-calcul, un système formel où tout est fonction, comme c'est le cas dans le paradigme que nous étudions ici.
Par exemple, pour créer « à la volée » la fonction $x \mapsto 2x + 1$, le code est le suivant :

```
lambda x: 2*x+1
```

On peut le lire ainsi : « une fonction qui prend `x` en argument et retourne `2*x+1` ».
On peut appliquer la fonction ci-dessus à un argument en entourant la fonction et son argument de parenthèses :

```
>>> (lambda x: 2*x+1)(5)
11
```

Une fonction `lambda` étant une expression, elle peut être nommée. Par conséquent, vous pouvez écrire le code précédent comme suit :

```
>>> f = lambda x: 2*x + 1
>>> f(5)
11
```

La fonction lambda ci-dessus équivaut à écrire ceci :

```
def f(x):
    return 2*x + 1
```

Les fonctions qui prennent plus d'un argument sont exprimées en Python en listant les arguments et en les séparant par une virgule ',' mais sans les entourer de parenthèses :

```
lambda x, y: x + y
```

c. Fonctions renvoyées comme résultats

En programmation fonctionnelle, les fonctions ne se contentent pas de pouvoir recevoir d'autres fonctions en argument, elles peuvent également renvoyer une fonction comme résultat.
Par exemple, pour calculer une approximation de la dérivée f' d'une fonction f, en utilisant le fait que $f'(x)$ est une valeur proche de du taux d'accroissement $(f(x+h) - f(x))/h$ pour une valeur de h assez petite, on peut écrire une fonction Python `derive` qui prend en paramètre une fonction `f` et *retourne une fonction* qui approche la dérivée de `f` :

```
def derive(f):
    h = 1e-7
    return lambda x: (f(x+h) - f(x))/h
```

On peut constater que cela fonctionne assez bien sur une fonction dont on connaît la dérivée :

```
>>> f = lambda x: x*x
>>> df = derive(f)
>>> df(3)
6.000000087880153
```

Un autre exemple est la fonction suivante qui crée une famille de fonctions qui ajoutent une valeur fixée à un nombre donné. La valeur fixée ajoutée est donnée en argument à la fonction.

```
def adder(x):
    return lambda y: x + y
```

```
>>> add_15 = adder(15)
>>> add_15(10)
25
```

3. Les fonctions `map`, `filter` et `reduce`

Les fonctions `map`, `filter` et `reduce` sont les fondements de la programmation fonctionnelle.

a. La fonction `map`

La fonction `map` permet d'appliquer la même fonction à tous les éléments d'une liste. Elle a deux paramètres : une fonction et une liste. Elle retourne un itérable contenant les images par la fonction des éléments de la liste initiale.
Par exemple, on peut obtenir la liste des longueurs d'une liste de mots :

```
>>> mots = ['banane','oeuf','poisson']
>>> list(map(len, mots))
[6, 4, 7]
```

ou les carrés des éléments d'une liste :

```
>>> list(map(lambda x: x * x, [2, 4, 5]))
[4, 16, 25]
```

b. La fonction `filter`

La fonction `filter` permet de sélectionner les éléments d'une liste vérifiant une condition donnée. Elle a deux paramètres : une fonction qui retourne un booléen (`True` ou `False`) et une liste. Elle retourne un itérable contenant les éléments de la liste initiale qui ont pour image `True`.
Par exemple, on peut obtenir la liste des éléments pairs d'une liste de nombre :

```
>>> est_pair = lambda x: x % 2 == 0
>>> list(filter(est_pair, [3, 4, 6, 12, 53]))
[4, 6, 12]
```

ou les éléments positifs d'une liste de nombres :

```
>>> nombres = [23, -7, 5, 42, -12]
>>> list(filter(lambda x : x > 0, nombres))
[23, 5, 42]
```

c. La fonction `reduce`

Contrairement à `map` et `filter` qui sont des fonctions intégrées à Python, pour utiliser `reduce`, vous devez l'importer depuis un module appelé `functools` :

```
from functools import reduce
```

`reduce` applique une fonction de deux arguments de manière cumulative aux éléments d'une liste, de gauche à droite. Exécuter l'instruction `help(reduce)` pour une description plus complète.
Par exemple, on peut obtenir le produit des éléments d'une liste de nombre :

```
>>> reduce((lambda x, y : x * y ), [1, 2, 3, 4])
24
```

La fonction donnée en argument dans la fonction `reduce` peut être vue de la manière suivante :

- Son premier argument est un accumulateur (des résultats intermédiaires) ;
- Son second argument est un élément de la liste (le suivant à chaque étape) ;
- La valeur retournée par la fonction est du même type que le premier argument (puisqu'elle est passée comme premier argument à la fonction avec la valeur suivante dans la liste).

Dans certains cas, il est nécessaire de donner une valeur initiale à cet accumulateur, qui n'est alors pas une valeur de la liste. On donne cette valeur comme troisième argument à la fonction `reduce`. Cela permet également de donner la valeur retournée par défaut, notamment quand la liste est vide.
Par exemple :

```
>>> reduce((lambda x, y : x + y ), [1, 2, 3, 4], -10)
0
```

On a ajouté à -10 tous les éléments de la liste `[1, 2, 3, 4]`.

VIII. Programmation dynamique

La programmation dynamique est une méthode algorithmique pour résoudre des problèmes d'optimisation. Elle consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires.
On peut dire que la résolution algorithmique d'un problème relève de la programmation dynamique si :

- le problème peut être résolu à partir de sous-problèmes similaires mais plus petits ;
- la solution optimale au problème posé s'obtient à partir des solutions optimales des sous-problèmes ;
- les sous-problèmes ne sont pas indépendants et un traitement récursif fait apparaître les mêmes sous-problèmes un grand nombre de fois.

En général, trouver une équation récursive du problème constitue une étape essentielle de sa résolution. L'algorithme obtenu est alors itératif et non plus récursif, avec une efficacité bien meilleure.
Nous traiterons quelques exemples d'application de cette méthode algorithmique au travers d'activités sur fichiers notebook.
On pourra lire la page Wikipédia concernant cette méthode.

IX. Recherche textuelle

Les algorithmes qui permettent de trouver une sous-chaîne de caractères dans une chaîne de caractères plus grande sont des « grands classiques » de l'algorithmique. On parle aussi de recherche d'un motif (au lieu de sous-chaîne) dans un texte.
Les algorithmes de recherche textuelle sont notamment utilisés en bio-informatique, en particulier lorsque l'on considère les brins d'ADN comme du texte utilisant les lettres A, C, G et T.
Après quelques exercices de manipulation de chaînes de caractère, nous traiterons quelques exemples de recherche textuelle.
Une activité, notamment, se concentrera sur l'**algorithme de Boyer-Moore**, particulièrement efficace, dont l'idée repose sur une recherche par la fin (plutôt que par le début) dans le texte, et de sauts dans le texte dépendants des occurrences des lettres dans la chaîne recherchée, ce qui nécessite un pré-traitement de la chaîne recherchée (plutôt que sur le texte dans lequel elle est cherchée).
On pourra voir plus d'éléments sur l'algorithme de Boyer-Moore sur la page Wikipédia.