

Algorithmes et Pensée Computationnelle

Fonctions, listes et dictionnaires

1 Les fonctions (Java ou Python) (30 minutes)

1.1 Rôle

Les `fonctions` permettent d'enregistrer du code dans une variable afin de réutiliser celui-ci à plusieurs endroits, et ainsi éviter de devoir le réécrire. Celles-ci sont définies une fois et peuvent être réutilisées autant de fois que l'on veut par la suite.

Les `fonctions` permettent de *factoriser* le code, offrant ainsi une structure plus générale à celui-ci et le rendant plus facilement modifiable.

Un exemple simple de fonctions est de créer une fonction qui imprime `"Hello World"` sur l'écran en l'appelant.

1.2 Syntaxe

1.2.1 Python

Pour définir une fonction, nous utilisons le mot `def` suivi du nom de notre `fonction`, puis des parenthèses `()`. Ces parenthèses peuvent contenir ou non des noms d'`arguments`, mais nous reviendrons dessus plus tard. Pour appeler une fonction, il suffit d'écrire son nom suivi de parenthèses `()`.

Pour reprendre l'exemple mentionné ci-dessus, nous déclarons une fonction du nom `print_hello` qui a pour unique utilité de `print` la phrase `"Hello World"`. Puis nous appelons cette fonction

```
def print_hello():  
    print("Hello World")
```

```
print_hello()
```

1.2.2 Java

En Java, les fonctions ont la structure suivante :

```
TypeDeRetour nomDeLaMethode() {  
    liste d'instructions  
}
```

Notez le typage dans les fonctions en Java. Si une fonction ne retourne rien, nous utilisons le type `void`. Une fonction dans Java doit être dans une classe, et pour exécuter le code de notre fonction, nous devons l'appeler dans la méthode `main`. L'exemple en Python ci-dessus peut être réécrit de la façon suivante :

```
public class Main {  
    public static void print_hello() {  
        System.out.println("Hello World!");  
    }  
    public static void main(String[] args) {  
        print_hello();  
    }  
}
```

1.3 Arguments

Comme dit précédemment, une fonction peut avoir un ou plusieurs `arguments`. Comme en maths, les arguments sont des valeurs que l'on passe à notre fonction et c'est avec ces valeurs que la fonction va effectuer ses opérations.

Par exemple en maths, lorsqu'on écrit $f(x) = x+2$, l'argument de la fonction f est x , je peux maintenant simplement écrire $f(2)$, ce qui signifie **remplacer x par 2 dans la fonction f** .

1.3.1 Python

En Python, les arguments fonctionnent de la même façon. Dans l'exemple suivant, nous créons une fonction du nom `print_name` qui prend un `argument` que nous appelons `name`. Nous nous servons de cet argument pour faire `print("Mon nom est", name)`. Nous appelons ensuite cette fonction avec un argument.

```
def print_name(name):  
    print("Mon nom est", name)  
  
print_name("Yasser")
```

1.3.2 Java

Nous reprenons l'exemple précédent et le réécrivons en Java :

```
public class Main {  
    public static void print_name(String name) {  
        System.out.println("Mon nom est" + name);  
    }  
    public static void main(String[] args){  
        print_name("Yasser");  
    }  
}
```

Question 1: (🕒 5 minutes) Python ou Java

Créez une fonction du nom de votre choix qui prend un argument `x` et qui `print(x+1)`.



Conseil

En Python, utilisez la fonction `print()` pour afficher du texte dans la console.
En Java, utilisez la fonction `System.out.println()`. N'oubliez pas de typer vos arguments !

Solutions :

1. Python :

```
def add_one(x):  
    print(x+1)  
  
add_one(5)
```

2. Java :

```
public class Main {  
    public static void add_one(int x) {  
        System.out.println(x+1);  
    }  
    public static void main(String[] args){  
        add_one(1);  
    }  
}
```

1.4 Return

Il est très commun que nous voulions enregistrer le résultat d'une fonction dans une variable. Il y a une différence notable entre `return` et `print`. En effet, `print` permet d'afficher du contenu, alors que `return` permet de renvoyer des résultats de fonctions. Par exemple, en maths, si nous avons une fonction $f(x) = x + 15$, nous pouvons faire $y = f(10)$ et nous savons donc que y vaut 25.

1.4.1 Python

En Python, si nous écrivons :

```
def f(x):  
    x + 15  
  
y = f(10)  
print(y)
```

Le `print(y)` va afficher `None`, car le résultat de `f(10)` ne vaut rien.

Pour résoudre ce problème, nous avons le mot-clef `return`, celui-ci permet de retourner une valeur de la fonction pour permettre d'enregistrer le résultat dans une variable. Pour reprendre l'exemple précédent :

```
def f(x):  
    return x + 15  
  
y = f(10)  
print(y)
```

Cette fois-ci `y` vaut bien `25`, car nous avons fait `return x + 15`.

1.4.2 Java

En Java, nous utilisons le mot-clef `return` aussi pour retourner le résultat. De plus, il faut spécifier le type de la variable que nous retournons.

```
TypeDeRetour nomDeLaMethode(type1 argument1) {  
    liste d'instructions  
    return variable;  
}
```

Nous convertissons le code Python ci-dessus en code Java :

```
public class Main {  
    public static int f(int x) {  
        return x + 15;  
    }  
    public static void main(String[] args){  
        int y = f(10);  
        System.out.println(y);  
    }  
}
```

Question 2: (🕒 5 minutes) Python ou Java

Écrivez une fonction de nom `f` qui prend un argument `x` et qui retourne `x * 10 + 2`, appelez cette fonction et enregistrer le résultat de celle-ci dans une variable `y`, puis `print y`.

💡 Conseil

Solutions :

1. Python :

```
def f(x):  
    return (x*10)+2  
y=f(2)  
print(y)
```

2. Java :

```
public class Main {  
    public static int f(int x) {  
        return (x * 10) + 2;  
    }  
    public static void main(String[] args){  
        int y = f(2);  
        System.out.println(y);  
    }  
}
```

1.5 Exercices d'applications (Optionnels)

Question 3: (🕒 5 minutes) Python ou Java

Complétez la fonction ci-dessous afin qu'elle retourne le cube de l'argument `x`. Imprimez le résultat de la fonction pour `x = 2`.

Python :

```
def cube(x):
    # Complétez ici

y = cube(2)
print(y)
```

Java :

```
public class Main {
    public static int cube(int x) {
        // Complétez ici
    }
    public static void main(String[] args){
        y = cube(2);
        System.out.println(y);
    }
}
```

Conseil

En Python, utilisez l'opérateur `**` afin de faire des puissances.
La librairie `java.lang.Math` peut vous être utile pour la version Java.

Solutions :

— **Python :**

```
def cube(x):
    return x**3

y = cube(2)
print(y)
```

— **Java :**

```
import java.lang.Math;

public class Main {
    public static int cube(int x) {
        return Math.pow(x, 3);
    }
    public static void main(String[] args){
        y = cube(2);
        System.out.println(y);
    }
}
```

Question 4: (🕒 5 minutes) **Python ou Java** Complétez la fonction ci-dessous afin qu'elle affiche les nombres de 0 à 10.

Python :

```
def counting():
    i = 0
    # Complétez ici

counting()
```

Java :

```
public class Main {
    public static void counting() {
        int i = 0;
        // Complétez ici
    }
    public static void main(String[] args){
        counting();
    }
}
```

Solutions :

— **Python :**

```
def counting():  
    i = 0  
    while i <= 10:  
        print(i)  
        i += 1
```

```
counting()
```

— **Java :**

```
public class Main {  
    public static void counting() {  
        int i = 0;  
        while (i <= 10){  
            System.out.println(i);  
            i += 1;  
        }  
    }  
    public static void main(String[] args){  
        counting();  
    }  
}
```

2 Listes et dictionnaires (Java ou Python)

2.1 Listes

2.1.1 Python

Les `listes` permettent de stocker plusieurs éléments, et sont immuables. Nous pouvons ainsi modifier leur contenu ou retirer et rajouter dynamiquement des éléments.

Méthodes principales :

- **Création** : `ma_liste = [1, 2, 3, 4]`
- **Modification** : `ma_liste[2] = 0`
- **Ajout** : `ma_liste.append(10)`
- **Suppression** : `ma_liste.pop()` pour enlever le dernier élément dans la liste ou `ma_liste.remove(10)` pour enlever 10 de la liste
- **Slicing / Indexation** : `my_list[0:2]` pour prendre les 2 premiers éléments de la liste [i (inclus) : j (exclu)]
- **Ajout avec indexation** : `my_list[0:2] = [4]` avec les éléments à rajouter entre crochets
- **Suppression avec indexation** : `my_list[0:2] = []`, sans rien entre les crochets

Question 5: (🕒 5 minutes) Comportement des listes en Python

Après l'exécution du code ci-dessous, à quoi va ressembler `my_list` ?

```
my_list = [1, 3, 5, 7, 11, 12]
print(my_list[0:3])

my_list.append(15)
my_list.pop()
my_list.remove(12)

my_list += [17, 30]
my_list[0:2] = [4]
my_list[0:3] = []
```

💡 Conseil

Écrivez le contenu de la liste après chaque opération pour pouvoir mieux comprendre le déroulement du programme.

Solutions :

```
[11, 17, 30]
```

2.1.2 Java

En Java, nous faisons la distinction entre `Array`, une liste à dimension fixe, et `ArrayList`, une liste à dimension variable.

1. **Array** : La taille de la liste doit être déclarée à l'initialisation, ou vous pouvez directement spécifier le contenu de la liste à l'initialisation. Après cela, la taille de la liste ne pourra être modifiée.

```
int[] mon_array = new int[5];
int[] mon_array1 = {1, 2, 3};
```

On utilise des accolades pour initialiser un `Array` avec des valeurs. Méthodes principales :

- (a) **Accès** : `ma_liste[0]`
 - (b) **Modification** : `ma_liste[1] = 10`
 - (c) **Slicing / Indexation** : `int[] newArray = Arrays.copyOfRange(oldArray, startIndex, endIndex);`
2. **ArrayList** : Plusieurs options s'offrent à nous quant à l'initialisation d'une `ArrayList`.

```
import java.util.ArrayList;
import java.util.List;

1. ArrayList liste = new ArrayList();
2. List<Integer> nombres = new ArrayList<>(6); // Dimension = 6

3. Collection elements = ...;
   List<Integer> nombres = new ArrayList<>(elements);
   // ArrayList contenant la collection elements
```

La première méthode est **déconseillée**, car elle ne spécifie pas explicitement le type des valeurs contenue dans l'`ArrayList`. Ainsi, nous devons toujours spécifier clairement le type pendant l'initialisation :

```
List<TypeDesValeurs> nomDeLaListe = new ArrayList<>();
```

Les méthodes principales pour les `ArrayList` :

- **Accès** : `ma_liste.get(0);`
- **Modification** : `ma_liste.set(index, value);`
- **Ajout** : `ma_liste.add(10);`
- **Suppression** : `ma_liste.remove("Java");` pour enlever "Java" de la liste ou `ma_liste.remove(10);` pour enlever l'élément à l'index 10.
- **Slicing / Indexation** : `ma_liste.sublist(startIndex, endIndex);`

2.2 Dictionnaires

Les `dictionnaires` sont des listes associatives, c'est-à-dire des listes qui relient une valeur à une autre.

2.2.1 Python

Dans un dictionnaire Python, on parle d'une relation **clef**, **valeur**. La **clef** étant le moyen de "*retrouver*" notre **valeur** dans notre **dictionnaire**.

Méthodes principales :

- Ajout de la **clef** "**Clef**" avec valeur "**Valeur**" : `my_dict["Clef"] = "Valeur"`
- Suppression d'une relation **clef-valeur** :
 - `my_dict.pop("Clef", None)` au cas où on sait pas si la **clef** en question est présente dans le dictionnaire
 - `del my_dict["Clef"]` si vous êtes sûrs que la **clef** en question est dans le dictionnaire

Vous pouvez trouver ci-dessous un exemple d'utilisation des dictionnaires :

```
annuaire = {"Shioban": 111, "Tyson": 222, "Shawn": 333 }
annuaire["Steven"] = 444
del annuaire["Tyson"]
print(annuaire.keys())
print(annuaire.values())
```

2.2.2 Java

En Java, les dictionnaires sont appelés des **Map**. La structure pour initialiser une **Map** est la suivante :

```
Map<Type de la clef, Type des éléments> dictionnaire = new HashMap<>();
```

HashMap est une des implémentations de **Map** les plus utilisées, et une des plus performantes.

Méthodes principales :

- Initialisation d'un dictionnaire avec comme **clef** des **String** et comme valeurs des **Integer** : `Map<String, Integer> age = new HashMap<>();`
- Ajout de la **clef** "**Justine**" avec valeur 22 : `age.put("Justine", 22)`
- Accès : `age.get("Justine")`
- Suppression d'une relation **clef-valeur** : `age.remove("Justine")`

3 Récursion

Le but principal de la récursion est de résoudre un gros problème en le divisant en plusieurs petites parties à résoudre.

Pour vous donner une idée de ce qu'est la récursion, pensez au travail du facteur. Chaque matin, il doit délivrer le courrier à plusieurs maisons. Il a certainement une liste de toutes les maisons du quartier par où il doit passer dans l'ordre. Par conséquent, il se rend devant une maison, pose le courrier puis va à la prochaine maison figurant sur sa liste. Ce problème est itératif car nous pouvons l'exprimer avec la boucle `for` : Pour chaque maison de sa liste, le facteur dépose le courrier.

```
maisons = ["A", "B", "C", "D"]

def delivrer_courrier_iteratively():
    for maison in maisons:
        print("Courrier délivré à la maison ", maison)

delivrer_courrier_iteratively()
```

Maintenant, imaginons que des stagiaires viennent aider le facteur à délivrer le courrier. Par conséquent, le facteur peut diviser son travail entre ses stagiaires. Pour ce faire, il attribue tout le travail de livraisons à un seul stagiaire qui doit déléguer son travail à deux autres stagiaires. Ces deux autres stagiaires ayant deux maisons à délivrer peuvent également déléguer leur travail à deux autres nouveaux stagiaires. Ces derniers, n'ayant chacun qu'une seule maison à délivrer doivent effectuer cette tâche chacun de leur côté. Ainsi, le facteur a reçu l'aide de 7 stagiaires : 3 délégateurs et 4 travailleurs.

Vous pensez certainement que cette manière de réfléchir est bizarre car vous auriez directement pensé que chaque stagiaire devra délivrer le courrier à une des 4 maisons de la liste. Cependant, ne connaissant pas le nombre de stagiaires travailleurs nécessaires, il est plus simple de commencer par un délégué et continuer à ajouter des délégués jusqu'à ce qu'il ne reste plus que la tâche à faire.

L'algorithme récursif suivant donne le même résultat que la fonction `delivrer_courrier_iteratively`, mais est un peu plus rapide. En effet, le courrier est livré plus vite à chaque maison.

```
maisons = ["A", "B", "C", "D"]

def delivrer_courrier_recursively(maisons):
    # Stagiaire travailleur livrant
    if len(maisons) == 1:
        maison = maisons[0]
        print("Courrier livré à la maison", maison)

    # Stagiaire délégué divisant sa tâche en deux
    else:
        mid = len(maisons) // 2
        first_half = maisons[:mid]
        second_half = maisons[mid:]

        # Stagiaire délégué déléguant ses deux parties
        # de tâche à deux autres stagiaires
        delivrer_courrier_recursively(first_half)
        delivrer_courrier_recursively(second_half)

delivrer_courrier_recursively(maisons)
```

3.1 Différence entre Boucle et Récursion

Une boucle `for` sert principalement à itérer des séquences de données pour les analyser ou manipuler. Par séquence, on entend un string, une liste, un tuple, un dictionnaire ou autre. En d'autres termes, une boucle passe d'une donnée à l'autre et effectue une opération sur chaque donnée. Ainsi, la boucle `for` se termine à la fin de la séquence.

Maintenant, une fonction récursive peut faire la même chose mais de manière plus efficace pour les plus grandes données. La principale différence entre une boucle et une fonction récursive est la façon dont elles se terminent. Une boucle s'arrête généralement à la fin d'une séquence alors qu'une fonction récursive s'arrête dès que la "base condition" est vraie.

Le but est que la fonction récursive se rappelle à chaque fois avec de nouveaux arguments ou qu'elle retourne une valeur finale.