Les fonctions

Table des matières

Introduction

Dans la plupart des programmes, des opérations similaires sont réalisées à plusieurs reprises. Nous avons vu le cas des boucles, mais il ne s'agit pas seulement de ça. Imaginons un site marchand, qui affiche les prix avec et sans les taxes, comme la TVA. Lorsque l'on ajoute un produit au catalogue, il sera nécessaire de calculer les taxes pour celui-ci. Si ensuite le produit est consulté par un client, le site devra calculer les taxes correspondant au pays de la personne qui consulte le produit, pour lui afficher. Le calcul des taxes est donc une opération réalisée à plusieurs reprises dans ce code, mais pas forcément à la suite, comme c'est le cas dans une boucle.

Pour cela les langages de programmations permettent de créer des fonctions, qui sont des bouts de code que l'on peut réutiliser à différents endroits. Ainsi un programme peut être vu comme un assemblage de briques, les fonctions, à la manière d'une construction en Lego.

I Factoriser le code avec les fonctions

Objectifs

- Comprendre la notion de factorisation de code ;
- Découvrir la notion de fonction.

Mise en situation

Dans son activité quotidienne, un développeur doit souvent réutiliser certains bouts de code car certaines parties d'algorithmes se retrouvent dans plusieurs programmes. Pour éviter de recopier systématiquement les mêmes instructions, il faut avoir recours à des fonctions. On dit qu'il **factorise** son code, c'est à dire qu'il supprime les duplications en les regroupant dans une structure : la **fonction**. Une fonction peut-être ensuite appelée à différents moments lors de l'exécution du programme, pour réaliser les mêmes opérations.

Recopie de code

Un développeur est souvent amené à réutiliser un bout de code à plusieurs endroits dans un programme ou dans différents programmes. Ces bouts de codes peuvent être des opérations de base tels que calculer le prix TTC à partir du prix HT ; ou alors des opérations plus complexes comme envoyer un mail. Par exemple, un site de e-commerce doit calculer le prix HT pour chaque produit et doit également envoyer des mails pour diverses raisons (inscription, message privé, information sur la commande, etc.).

Problème de gestion des recopies

⚠ Attention

La recopie pose deux problèmes majeurs. Le premier est que sur des projets très importants, le nombre de recopies d'un bout de code donné pourrait être très grand. En évitant cette recopie, l'espace disque occupé par le code pourrait être significativement réduit. Le second problème est la gestion des versions. En effet, si le développeur découvre que le code recopié possède un bug ou pourrait être optimisé, il va devoir modifier l'ensemble des copies. Pour faire simple, la recopie crée un problème de **versionnage** qui est à éviter à tout prix particulièrement sur des projets à la taille importante, car pouvant provoquer des erreurs difficiles à diagnostiquer.

Factoriser son code grâce à des fonctions

Méthode

La factorisation du code est souvent réalisée grâce à des fonctions. Une fonction contient un ensemble d'instructions ré-utilisables que le programme exécutera à chaque **appel**, plutôt que recopier le code. Voici en JavaScript puis en Python comment déclarer une fonction puis l'appeler.

```
1 /** JavaScript : affiche 'Hello world!' */
2 function helloWorld () {
3   console.log('Hello world!')
4   console.log('I am a developper.')
5 }
6
7 helloWorld() // On appelle la fonction une première fois
```

```
8
9 // D'autres opérations
10
11 helloWorld() // Second appel de la fonction
12
```

```
1 """Python : affiche affiche 'Hello world!' """
2 def hello_world():
3   print("Hello world!")
4   print("I am a developper.")
5
6 hello_world()  # On appelle la fonction une première fois
7
8 # D'autres opérations
9
10 hello_world()  # Second appel de la fonction
```

Chantons Sloubi

Exemple

Voici un exemple trivial en JavaScript dans lequel on réutilise plusieurs fois le même bout de code. Dans le second programme, on définit préalablement une fonction et celle-ci sera appelée pour éviter de recopier le code.

```
1 /** JavaScript : chante sloubi */
2 console.log('Une partie de Sloubi')
3
4 for (let i = 0; i < 20; i++) {
5   console.log('Sloubi', i)
6 }
7
8 console.log('Allez une autre')
9 for (let i = 0; i < 20; i++) {
10   console.log('Sloubi', i)
11 }
12
13 console.log('Encore!')
14 for (let i = 0; i < 20; i++) {
15   console.log('Sloubi', i)
16 }
17
18 console.log('Une dernière.')
19 for (let i = 0; i < 20; i++) {
20   console.log('Sloubi', i)
21 }
22
23 console.log('Une pour la route')
24 for (let i = 0; i < 20; i++) {
25   console.log('Sloubi', i)
26 }
27
```

```
1 /** JavaScript : chante sloubi avec moins de code */
2 function sloubi () {
3   for (let i = 0; i < 20; i++) {
4     console.log('Sloubi', i)
5   }
6 }
7
8 console.log('Une partie de Sloubi')
9 sloubi()
10
11 console.log('Allez une autre')
```

```
12 sloubi()
13
14 console.log('Encore!')
15 sloubi()
16
17 console.log('Une dernière.')
18 sloubi()
19
20 console.log('Une pour la route')
21 sloubi()
22
```

À retenir

- Un développeur est souvent amené à réutiliser des morceaux de code qu'il a déjà développé.
- Les fonctions permettent notamment de factoriser le code et d'éviter une recopie.

II Exercice : Appliquer la notion

Question

[solution n°1 p. 25]

Voici un code avec des recopies inutiles.

Définir une fonction et donner la version factorisée de ce programme.

```
let choice = ['Pierre', 'Feuille', 'Ciseaux']

console.log('Une petite partie?')
let index = Math.floor(Math.random() * choice.length)
console.log(choice[index])

console.log('Une autre?')
index = Math.floor(Math.random() * choice.length)
console.log(choice[index])

console.log('Une dernière!')
index = Math.floor(Math.random() * choice.length)
console.log(choice[index])

```

Indice :

L'ensemble du code **dupliqué** doit être factorisé à l'intérieur d'une unique fonction.

III Usage des fonctions

Objectifs

- Savoir rendre les fonctions plus génériques grâce aux arguments ;
- Savoir renvoyer une information depuis une fonction.

Mise en situation

Vous avez sans doute appris au collègue le concept de fonction mathématique. C'est une formule de calcul qui permet de calculer une valeur en fonction de différents paramètres d'entrée. Et bien les fonctions en informatique fonctionnent de manière très similaire. Mais pas de panique, il n'y a pas besoin d'être un expert en maths.

Une fonction est composée d'un ensemble d'instructions qui réalisent une opération quelconque. Ces instructions peuvent se baser sur des paramètres d'entrées de la fonction, c'est à dire des variables, et peuvent produire un résultat en retour, donc une nouvelle variable.

Fonction — Fondamental

Une fonction est un ensemble d'instructions (aussi appelé sous-programme) qui a pour but de retourner un résultat ou de réaliser une tâche donnée. Cette suite d'instruction peut être appelée à plusieurs endroits du code grâce à son nom. À chaque appel de la fonction, l'ensemble de ses instructions est exécuté par l'ordinateur.

Paramètre — Az Définition

Une fonction peut avoir un ou plusieurs paramètres. Ces paramètres sont équivalents aux antécédents pour les fonctions mathématiques. Un paramètre est une valeur d'un type quelconque et qui sera utilisée par la fonction pour ses calculs. À noter qu'il est possible d'avoir un type différent pour chaque argument d'une fonction.

Valeur de retour — Az Définition

Une fonction peut retourner ou non un résultat. Cette valeur est équivalente à l'image d'une fonction mathématique. Une fonction peut retourner tout type de valeurs, que ce soit un type simple (entier, chaîne de caractères, etc.) ou un type complexe (tableau, enregistrement, etc.).

Procédures et fonctions — Complément

Certains informaticiens puristes distinguent les notions de procédures et de fonctions. Le seul élément différenciant ces deux notions est le fait qu'une fonction retourne systématiquement une valeur alors qu'une procédure non.

Retourner le maximum d'une liste

Exemple

```javascript
/** JavaScript : fonction qui retourne le maximum d'une liste d'entiers. */
function maxList (localList) {
  let max = localList[0]
  for (let i = 0; i < localList.length; i++) {
    if (max < localList[i]) {
      max = localList[i]
    }
  }
  return max
}

console.log(maxList([2, 5, 10, 3]))

```

```python
"""Python : fonction qui retourne le maximum d'une liste d'entiers. """
def max_list(local_list):
  max = local_list[0]
  for i in range(1, len(local_list)):
    if max < local_list[i]:
      max = local_list[i]

  return max

print(max_list([2, 5, 10, 3]))
```

Modulariser son code

Méthode

Par défaut, il est intéressant de créer des fonctions car cela va permettre d'avoir un code fonctionnant par modules. Le programme principal deviendra plus lisible car sa taille sera réduite. Utiliser par défaut des fonctions même en l'absence de recopie réduira donc la taille du programme principal et réduira le travail du développeur, si par la suite il avait à nouveau besoin de cette fonction sans que cela ait été prévu.

Un développeur peut alors séparer son programme en plusieurs fichiers :

- Un fichier par thématique regroupant les fonctions autour d'un même thème (calcul, interaction, etc.),
- Un fichier principal qui appellera les différentes fonctions.

Une telle séparation aidera la relecture et la reprise du projet par un développeur tiers. Cela aidera également le développeur lui-même à découper et structurer sa tâche.

À retenir

- Les fonctions sont une suite d'instructions qui réalise un calcul en s'appuyant sur des paramètres.
- Une fonction peut retourner une valeur, correspondant par exemple au résultat d'un calcul.

IV Exercice : Appliquer la notion

Question 1

[solution n°2 p. 25]

Écrire une fonction `htToTtc` qui prend en paramètre une liste de prix hors taxes et qui retourne la liste des prix TTC. Il faut ajouter 20 % à un prix hors taxes pour avoir le prix TTC.

Indice :

La syntaxe pour créer une fonction en JavaScript est la suivante :

```
function nomFonction(param1, param2, ...) {
  // Instructions de la fonction
}
```

Indice :

```
function htToTtc (htPriceList) {
  const ttcPriceList = []
  for (let i = 0; i < htPriceList.length; i++) {
    ...
  }
}

```

Indice :

Calcul du prix TTC pour une valeur : `ttcPrice = htPrice * 1.2`

Indice :

```
function htToTtc (htPriceList) {
  const ttcPriceList = []
  for (let i = 0; i < htPriceList.length; i++) {
    const ttcPrice = htPriceList[i] * 1.2
    ttcPriceList.push(ttcPrice)
  }
  ...
}

```

Indice :

L'instruction pour retourner une valeur depuis une fonction est la suivante :

```
return nomVariable
```

Question 2

[solution n°3 p. 25]

Que retourne la fonction lorsqu'on envoie `[2, 5, 10]` comme paramètre ?

Indice :

```
function htToTtc (htPriceList) {
  ...
}

console.log(...)
```

Indice :

```
function htToTtc (htPriceList) {
  ...
}

console.log(htToTtc(...))
```

V Fonctions et portée des variables

Objectif

- Comprendre les contraintes de portées des variables pour les fonctions.

Mise en situation

Nous connaissons bien le concept de portée d'une variable. Elle correspond à la zone du code source dans laquelle la variable est déclarée, et donc utilisable. Par défaut, les variables sont locales et ne peuvent être utilisées que dans le bloc de code où elles ont été déclarées.

Les fonctions étant des blocs de code comme les autres, les contraintes de portée des variables s'y appliquent de la même manière. Une variable définie dans une fonction n'est valable que dans cette fonction.

Portée des variables

Rappel

Lorsqu'on définit une variable, elle est associée au bloc où elle se déclarée et n'est visible que dans celui-ci. On parle ici de variable locale.

Portée dans une fonction

Fondamental

Une variable définie dans une fonction ne sera accessible qu'à l'intérieur de celle-ci. En Python ce bloc est distinguable grâce à l'indentation et en JavaScript grâce aux accolades.

Portée des variables locales en JavaScript

Exemple

```
1 /** JavaScript : function retournant le maximum d'une liste */
2 function maxList (localList) {
3   let max = localList[0] // La variable max n'existe que durant l'exécution de
  la fonction
4   for (let i = 0; i < localList.length; i++) {
5     if (max < localList[i]) {
6       max = localList[i]
7     }
8   }
9   return max
10 }
11
12 console.log('Le maximum de la liste est ', maxList([2, 5, 10, 3]))
```

Même nom, portée différente

Attention

Il est possible définir dans une fonction une variable dont le nom est le même que celui d'une variable dans le programme qui l'appelle.

Si les deux variables sont locales à deux blocs différents, elles existent indépendamment et leurs valeurs ne sont pas conflictuelles.

```javascript
/** JavaScript : function retournant le maximum d'une liste */
function maxList (localList) {
  let max = localList[0]
  for (let i = 0; i < localList.length; i++) {
    if (max < localList[i]) {
      max = localList[i]
    }
  }
  return max
}

let max = maxList([2, 5, 10, 3]) // Cette variable max est différente de la variable max utilisée dans la fonction
console.log('Le maximum de la liste est ', max)

```

Rappel

En JavaScript, il est possible de définir une variable soit directement sans préfixe, soit avec `var` soit avec `let`.

Les syntaxes sans préfixe et avec `var` permettent de définir une variable qui persistera une fois le bloc fini, donc une **variable globale**. On évite d'utiliser les variables globales.

La syntaxe `let` permet de définir une variable qui sera détruite une fois le bloc terminé.

Déclaration des variables dans les fonctions

Méthode

Il faut toujours utiliser `let` ou `const` dans les fonctions JavaScript.

Effets de bord

Attention

Ne pas utiliser `let` pour définir une variable dans une fonction conduit à des **effets de bord**, il faut donc **systématiquement** définir les variables utilisées dans une fonctions pour les éviter.

En effet, une variable définie sans indication supplémentaire écrase les variables définies dans le code appelant.

Voici un exemple où une variable est modifiée sans que cela soit voulu.

La première version renvoie un résultat erroné. La seconde version corrige ce bug en ajoutant un `let` dans la première ligne de la fonction `max_liste`.

```javascript
/** JavaScript : code ayant un problème avec la portée d'une de ses variables */
function maxList (localList) {
  // Sans let, on déclare une variable globale qui écrase la variable max existante
  max = localList[0]
  for (let i = 0; i < localList.length; i++) {
    if (max < localList[i]) {
      max = localList[i]
    }
  }
  return max
}

// On cherche le maximum d'une liste de listes d'entiers positifs
```

```
const listception = [
  [1, 12, 5],
  [4, 3, 10],
  [2, 7, 1, 9]
]
let max = 0

for (let i = 0; i < listception.length; i++) {
  const tempMax = maxList(listception[i])
  if (max < tempMax) {
    max = tempMax
  }
}

console.log(max) // Affichera 9 alors que le maximum est 12
```

```
/** JavaScript : code corrigeant l'effet de bord dû à la portée d'une variable
 */
function maxList (localList) {
  // Avec let, on déclare une variable locale à la fonction
  let max = localList[0]
  for (let i = 0; i < localList.length; i++) {
    if (max < localList[i]) {
      max = localList[i]
    }
  }
  return max
}

// On cherche le maximum d'une liste de listes d'entiers positifs
const listception = [
  [1, 12, 5],
  [4, 3, 10],
  [2, 7, 1, 9]
]
let max = 0

for (let i = 0; i < listception.length; i++) {
  const tempMax = maxList(listception[i])
  if (max < tempMax) {
    max = tempMax
  }
}

console.log(max) // Affichera la bonne valeur
```

À retenir

- Les contraintes de portée de variables sont valables également pour les fonctions.
- En JavaScript, il est indispensable d'utiliser `let` pour définir des variables dans des fonctions pour éviter les effets de bord.

VI Exercice : Appliquer la notion

Voici un code qui ne retourne pas le résultat prévu :

```
function htToTtc (htPrice) {
  priceToPay = 1.2 * htPrice
  return priceToPay
}

const prices = [10, 20, 5]

let priceToPay = 0
for (let i = 0; i < prices.length; i++) {
  const currentPrice = htToTtc(prices[i])
  priceToPay = priceToPay + currentPrice
}

console.log('Vous devez payer ', priceToPay)

```

Question 1

[solution n°4 p. 25]

Exécuter le code et donner le résultat affiché.

Question 2

[solution n°5 p. 26]

Quel résultat devrait être retourné ?

Indice :

Pour passer du prix TTC au prix HT, il faut ajouter 20 % au prix hors taxes.

Question 3

[solution n°6 p. 26]

Corriger le bug et proposer le code corrigé.

Indice :

Il y a un problème lié à la portée d'une des variables.

VII Fonctions pré-définies

Objectif

- Découvrir les fonctions pré-définies.

Mise en situation

Pour faciliter la vie du développeur, les langages intègrent des fonctions pré-définies répondant à des usages très fréquents. Ainsi, les développeurs n'ont pas à ré-implémenter systématiquement ces fonctions de base. C'est le cas par exemple des fonctions de manipulation de tableau, comme le tri.

Fonctions pré-définies par les langages

Fondamental

Une fonction pré-définie est une fonction disponible **nativement** dans un langage de programmation. Un développeur peut avoir recours à cette fonction sans importer de bibliothèques de fonctions. Les fonctions pré-définies répondent la plupart du temps à des usages très simples : affichage, tri de liste, etc.

Fonctions pré-définies les plus utilisées

Rappel

Voici trois des fonctions pré-définies les plus utilisées par les développeurs :

- **Fonctions de cast** : ces fonctions transforment le type d'une variable. Par exemple, pour transformer une variable en un nombre il faut utiliser `Number()` en JavaScript et `float()` en Python. Le cast est très utilisé par les développeurs lorsqu'il faut transformer l'entrée de l'utilisateur. L'entrée de l'utilisateur étant toujours une chaîne de caractères, si on souhaite récupérer un prix, il faut caster l'entrée de l'utilisateur pour la transformer en nombre.
- **Fonctions d'affichage** : ces fonctions affichent des informations dans la console notamment. En JavaScript, il est possible d'utiliser `console.log()` et en Python, `print()`.
- **Fonction de tri** : cette fonction trient une liste donnée. En Python comme en JavaScript, il existe la fonction `sort()`.

Utilisation de fonctions pré-définies

Exemple

Voici un exemple dans lequel une liste de chaînes de caractères est transformée en liste d'entiers triée :

```
/** JavaScript : utilisation de fonctions pré-définies */
const priceList = ['12.4', '17.1', '5.0']
const newList = []

for (let i = 0; i < priceList.length; i++) {
  newList.push(Number(priceList[i]))
}

```

```
newList.sort((a, b) => a - b) // Trie numériquement dans l'ordre croissant

console.log(newList)
```

```
"""Python : utilisation de fonctions pré-définies """
price_list = [ "12.4", "17.1", "5.0"]
new_list = []

for i in range(len(price_list)):
  new_list.append(float(price_list[i]))
new_list.sort()

print(new_list)
```

Découvrir les fonctions pré-définies

Complément

Pour trouver découvrir l'ensemble des fonctions pré-définies, il est nécessaire de regarder la documentation :

- En JavaScript : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux
- En Python : https://docs.python.org/fr/3.8/library/functions.html

Ne pas redéfinir une fonction prédéfinie

Attention

Il faut bien faire attention à ne pas écraser une fonction pré-définie en nommant une de ses fonctions avec le nom d'une des fonctions pré-définies.

À retenir

- Il existe des fonctions présentent nativement dans les langages qui visent à simplifier la vie du développeur.
- Elles assurent une optimisation maximale pour des tâches très simples.

VIII Exercice : Appliquer la notion

La fonction JS pré-définie `prompt()` affiche la chaîne de caractères fournie en paramètre et retourne ce que l'utilisateur entre dans la console. Lien vers la documentation : https://developer.mozilla.org/fr/docs/Web/API/Window/prompt.

```
1 const name = prompt("Entrer votre nom")
2 // Affiche le message et la variable nom prendra pour valeur ce que l'utilisateur a entré
```

Question 1

[solution n°7 p. 26]

Écrire un programme qui utilise la fonction `prompt()` pour demander la date de naissance de l'utilisateur et afficher ensuite : *« Vous êtes né.e le JJ/MM/AAAA »*.

Question 2

[solution n°8 p. 26]

Combien de fonctions pré-définies différentes sont utilisées dans ce programme ?

IX Quiz

Exercice 1 : Quiz - Culture

[solution n°9 p. 26]

Exercice

Quelles sont les raisons qui plaident en défaveur de la recopie de code ?

- [ ] **A** Copier le code induit un temps de développement allongé lors de la maintenance du code.
- **B** Copier le code induit un temps de développement allongé lors la création du code.
- **C** Copier le code augmente la taille des programmes.
- **D** Copier le code augmente les temps d'exécution.

Exercice

Lorsqu'on utilise une fonction pour éviter des recopies, on parle de :

- **A** Développement du code
- **B** Factorisation du code
- **C** Fonctionnement du code
- **D** Fonctionnarisation du code

Exercice

Quel élément différencie une fonction et une procédure ?

- **A** Une fonction accepte des paramètres.
- **B** Une fonction exécute des instructions.
- **C** Une fonction retourne un résultat d'exécution.

Exercice

Les types des paramètres passés à une fonction doivent être identiques ?

- **A** Vrai
- **B** Faux

Exercice 6 : Quiz - Méthode

[solution n°10 p. 28]

Exercice

Pour définir une variable locale dans une fonction, quel mot-clé faut-il utiliser ?

- **A** let
- **B** var
- **C** Aucune (juste le nom de la variable sans mot-clé le précédant).

Exercice

Cette fonction `deviner` pourrait causer un effet de bord à cause de l'usage du même nom de variable `nombre` dans le programme principal et dans la fonction :

```
1 function deviner (nombreADeviner) {
2   let nombre = Number(prompt("Donner un nombre"))
3   return nombre === nombreADeviner
4 }
5
6 let nombre = 3
7 while (!deviner(nombre)) {
8   console.log('Raté')
9 }
10
11 console.log('Bien joué')
12
```

- **A** Vrai
- **B** Faux

Exercice 9 : Quiz - Code

[solution n°11 p. 28]

Exercice

Quelle fonction pré-définie JavaScript permet de transformer une chaîne de caractères en nombre ?

- **A** float()
- **B** Number()

C Integer()

D Float()

Exercice

Quelle(s) syntaxe(s) permet(tent) de retourner plusieurs variables `date` et `message` depuis une fonction JavaScript ?

A `return date, message`

B `return [date, message]`

C `return date && return message`

D `return date return message`

X Exercice : Défi

Nous souhaitons créer un réseau social en JavaScript. Chaque membre du réseau est représenté par un pseudo (chaîne de caractère).

Voici un programme que nous allons factoriser pas à pas :

```
const friendList = []
let addFriendChoice = true
console.log('Ajoutons un premier ami')
while (addFriendChoice) {
  const pseudoFriend = prompt('Quel est son pseudo ?')
  friendList.push(pseudoFriend)

  addFriendChoice = prompt('Souhaitez-vous ajouter un autre ami? O ou N?') === 'O'
}

console.log('Retirons un ami de votre liste')
const friendToRemove = prompt('Quel est son nom?')
// A compléter

console.log("Voici votre liste d'amis:", friendList)

```

Question 1

[solution n°12 p. 29]

Compléter le code afin de supprimer l'ami demandé, s'il existe.

Indice :

Il suffit de parcourir la liste d'amis et de supprimer l'ami correspondant. On ne retournera pas d'erreur si la personne n'est pas trouvée.

Indice :

Pour supprimer le i-ème élément d'une liste, il faut utiliser l'instruction suivante :

```
friendList.splice(i, 1)
```

Indice :

Une fois l'ami trouvé et supprimé, il n'est plus nécessaire de continuer la recherche. L'instruction `break` permet de sortir d'une boucle.

Maintenant que notre programme est complet, nous allons le **modulariser**.

Question 2

[solution n°13 p. 29]

Proposer une fonction qui se chargera d'ajouter un ami. Elle prendra comme paramètre la liste d'amis et retournera la nouvelle version de la liste d'amis suite à l'ajout. La fonction s'appellera `addFriend`.

Indice :

Le corps de la fonction sera sensiblement similaire au code existant pour ajouter un ami. Il comportera en plus une instruction `return`.

Question 3

[solution n°14 p. 29]

Faire de même pour retirer un ami et on nommera la fonction `removeFriend`.

Question 4

[solution n°15 p. 30]

Les deux fonctions `addFriend` et `removeFriend` comportent une duplication : la partie du code qui demande les informations sur l'ami à ajouter ou à enlever est identique.

Créer une fonction sans paramètre :

- Qui demandera à l'utilisateur un pseudo
- Qui retournera la chaîne de caractères correspondante.

Cette fonction sera nommée `infoFriend()`.

Question 5

[solution n°16 p. 30]

Donner le programme complet avec les fonctions.

Indice :

Pour éviter tout effet de bord, il faut envoyer une copie d'une liste aux fonctions plutôt que la liste elle-même. Pour faire une copie d'une liste il faut utiliser l'instruction suivante :

```
1 liste.slice()
```

Indice :

N'oubliez pas de faire appel à la fonction `infoFriend()` dans les fonctions d'ajout et de suppression.

Conclusion

Les fonctions sont absolument essentielles en programmation, et sont utilisées dans la plupart des programmes. Elles permettent de factoriser du code qui se répète à plusieurs reprise, facilitant le travail du développeur. De plus cela rend le code plus fiable à maintenir, en évitant que des portions de programme dupliquées ne soient pas modifiées de la même manière partout.

Nous avons vu comment créer et utiliser des fonctions, et nous avons constaté que nous utilisions en fait des fonctions depuis nos début en programmation, avec les fonctions pré-définies dans les langages.

Solutions des exercices

Solution n°1

[exercice p. 7]

```
function play () {
  const choice = ['Pierre', 'Feuille', 'Ciseaux']
  const index = Math.floor(Math.random() * choice.length)
  console.log(choice[index])
}

console.log('Une petite partie?')
play()

console.log('Une autre?')
play()

console.log('Une dernière!')
play()

```

Solution n°2

[exercice p. 10]

```
function htToTtc (htPriceList) {
  const ttcPriceList = []
  for (let i = 0; i < htPriceList.length; i++) {
    const ttcPrice = htPriceList[i] * 1.2
    ttcPriceList.push(ttcPrice)
  }
  return ttcPriceList
}

```

Solution n°3

[exercice p. 10]

```
function htToTtc (htPriceList) {
  const ttcPriceList = []
  for (let i = 0; i < htPriceList.length; i++) {
    const ttcPrice = htPriceList[i] * 1.2
    ttcPriceList.push(ttcPrice)
  }
  return ttcPriceList
}

console.log(htToTtc([2, 5, 10]))
```

```
[ 2.4, 6, 12 ]
```

Solution n°4

[exercice p. 15]

12

Solution n°5

[exercice p. 15]

42

Solution n°6

[exercice p. 15]

```
function htToTtc (htPrice) {
  const priceToPay = 1.2 * htPrice
  return priceToPay
}

const prices = [10, 20, 5]

let priceToPay = 0
for (let i = 0; i < prices.length; i++) {
  const currentPrice = htToTtc(prices[i])
  priceToPay = priceToPay + currentPrice
}

console.log('Vous devez payer ', priceToPay)

```

Solution n°7

[exercice p. 18]

```
const day = prompt('Donner votre jour de naissance')
const month = prompt('Donner votre mois de naissance')
const year = prompt('Donne votre année de naissance')

console.log('Vous êtes né·e le', day, '/', month, '/', year)

```

Solution n°8

[exercice p. 18]

Il y en a deux : `prompt` et `console.log` .

Solution n°9

[exercice p. 19]

Exercice

Quelles sont les raisons qui plaident en défaveur de la recopie de code ?

A Copier le code induit un temps de développement allongé lors de la maintenance du code.

B Copier le code induit un temps de développement allongé lors la création du code. Lors de la création du code, effectuer un copier-coller n'est en général par très compliqué... c'est d'ailleurs ce qui fait qu'on est parfois tenté d'utiliser cette méthode plutôt que de factoriser le code.

C Copier le code augmente la taille des programmes.

D

Copier le code augmente les temps d'exécution.

Le temps d'exécution sera sensiblement équivalent que le code soit factorisé dans une fonction ou non.

Exercice

Lorsqu'on utilise une fonction pour éviter des recopies, on parle de :

A Développement du code

B Factorisation du code

C Fonctionnement du code

D Fonctionnarisation du code

Exercice

Quel élément différencie une fonction et une procédure ?

A Une fonction accepte des paramètres. Une procédure également.

B Une fonction exécute des instructions. Une procédure également.

C Une fonction retourne un résultat d'exécution.

Une fonction renvoie un résultat, ce n'est pas le cas pour une procédure.

Exercice

Les types des paramètres passés à une fonction doivent être identiques ?

A Vrai

B Faux

Chaque paramètre d'une fonction peuvent avoir un type différent.

Solution n°10

[exercice p. 20]

Exercice

Pour définir une variable locale dans une fonction, quel mot-clé faut-il utiliser ?

A let

B var

C Aucune (juste le nom de la variable sans mot-clé le précédant).

Exercice

Cette fonction `deviner` pourrait causer un effet de bord à cause de l'usage du même nom de variable `nombre` dans le programme principal et dans la fonction :

```
1  function deviner (nombreADeviner) {
2    let nombre = Number(prompt("Donner un nombre"))
3    return nombre === nombreADeviner
4  }
5
6  let nombre = 3
7  while (!deviner(nombre)) {
8    console.log('Raté')
9  }
10
11 console.log('Bien joué')
12
```

A Vrai

B Faux

La variable est redéfinie dans la fonction grâce au mot-clé `let`. Le définition globale sera mise de côté durant l'exécution de la fonction.

Solution n°11

[exercice p. 20]

Exercice

Quelle fonction pré-définie JavaScript permet de transformer une chaîne de caractères en nombre ?

A float() Cette fonction sert à transformer une chaîne en nombre, mais en Python

B Number()

C Integer()

D Float()

Exercice

Quelle(s) syntaxe(s) permet(tent) de retourner plusieurs variables `date` et `message` depuis une fonction JavaScript ?

A `return date, message`

B `return [date, message]`

C `return date && return message`

D `return date return message`

Une fonction JavaScript ne peut retourner qu'une seule valeur. Pour retourner plusieurs éléments, la solution est de les renvoyer au sein d'un tableau, qui forme bien une seule valeur.

Solution n°12

[exercice p. 22]

```
for (let i = 0; i < friendList.length; i++) {
  if (friendToRemove === friendList[i]) {
    friendList.splice(i, 1)
    break
  }
}
```

Solution n°13

[exercice p. 22]

```
function addFriend (friends) {
  const pseudoFriend = prompt('Quel est son pseudo ?')
  friends.push(pseudoFriend)
  return friends
}

```

Solution n°14

[exercice p. 23]

```
function removeFriend (friends) {
  const friendToRemove = prompt('Quel est son pseudo ?')
  for (let i = 0; i < friends.length; i++) {
    if (friendToRemove === friends[i]) {
      friends.splice(i, 1)
      break
    }
  }
  return friends
}

```

Solution n°15

[exercice p. 23]

```
function infoFriend () {
  const pseudoFriend = prompt('Quel est son pseudo ?')
  return pseudoFriend
}

```

Cette factorisation a un autre avantage : si on veut demander stocker plus d'informations sur les amis, la modification de la demande ne devra être faite qu'à un seul endroit.

Solution n°16

[exercice p. 23]

```
function infoFriend () {
  const pseudoFriend = prompt('Quel est son pseudo ?')
  return pseudoFriend
}

function addFriend (friends) {
  friends.push(infoFriend())
  return friends
}

function removeFriend (friends) {
  const friendToRemove = infoFriend()
  for (let i = 0; i < friends.length; i++) {
    if (JSON.stringify(friendToRemove) === JSON.stringify(friends[i])) {
      friends.splice(i, 1)
      break
    }
  }
  return friends
}

let friendList = []
let addFriendChoice = true
console.log('Ajoutons un premier ami')
while (addFriendChoice) {
  friendList = addFriend(friendList.slice())

  addFriendChoice = prompt('Souhaitez-vous ajouter un autre ami? O ou N?') === 'O'
}

console.log('Retirons un ami de votre liste')
friendList = removeFriend(friendList.slice())

console.log("Voici votre liste d'amis:", friendList)

```

Notez que si le code est plus long, il s'adaptera mieux aux changements. Pour faire évoluer le code et permettre de mettre à jour un ami, ou pour ajouter un ami à un autre endroit, il suffira de ré-utiliser les fonctions `addFriend` et `infoFriend`.

La **modularisation** du code est donc un processus un peu plus coûteux au départ, mais qui évite une factorisation pénible et plus longue par la suite.