CHAPITRE 9 : Langages et programmation

1 Constructions élémentaires dans un programme 2
1.1 Histoire 2
1.2 Constructions élémentaires dans un programme 3
2 Diversité et unité des langages de programmation 3
2.1 Paradigmes 3
2.2 Interprétation ou compilation 3
2.3 Différences de syntaxe 4
3 Spécification d'une fonction 5
3.1 Prototype d'une fonction 5
3.2 Documentation d'une fonction 5
3.3 Les assertions 6
4 Mise au point de programmes 7
5 Utilisation de bibliothèques 7
6 Exemple avec une bibliothèque et des assertions : Tri d'un tableau selon une colonne par la fonction sorted de Python 8

CHAPITRE 9 : Langages et programmation

1 Constructions élémentaires dans un programme

1.1 Histoire

Il existe plus 700 langages de programmation parmi lesquels Python, Java, JavaScript, C++, C# (C sharp).

1949 *Le langage assembleur*

C'est la première représentation textuelle du langage machine lisible par un humain. Cela reste un langage de bas niveau c'est à dire proche du langage machine (code binaire éventuellement lisible sous forme hexadécimale).

```
1  LDX #0
2  LDA #0
3  INX
4  ADC #3
5  CPX #2
6  BNE -7
```

1951 *Le premier compilateur*

Grace Hopper conçoit le premier programme "compilateur" c'est à dire un programme qui transforme un programme écrit en "**code source**" c'est à dire en langage de haut niveau (anglais) en un programme écrit en "**code objet**" c'est à dire en langage de bas niveau.

1964 *Le langage Basic*

BASIC signifie **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode. C'est le premier langage destiné au grand public. Les lignes sont numérotées et une des instructions est GOTO suivi du numéro de ligne. Cela permet de réaliser des boucles.

```
370 PRINT AF1$;"Impossible."
371 IF CP=1 OR CO=1 THEN GOTO 380
372 TEMPO=1.5:GOSUB 11100:GOTO 350
373 IF MCH=0 OR CO=1 THEN GOTO 366
374 PRINT:PRINT A$:CO=1
375 TEMPO=1.5:GOSUB 11100:GOTO 350
```

1972 *Le langage C*

C'est un langage de bas niveau. Chaque instruction est conçue pour être compilée en un nombre d'instructions machine assez prévisible en termes d'occupation mémoire et de charge de calcul. Il a été inventé au "Bell Labs" aux Etats-Unis. Le premier Unix était en assembleur. Unix a été réécrit en C en 1973.

```c
int main(void)
{
    printf("Exemple 2\n");
    fonction_1();
    fonction_2();
    printf("bye\n");
    return 0;
}
```

1972 Le paradigme[1] "*programmation orientée objet*".

L'objet est un conteneur symbolique qui contient des données et des comportements.

Par exemple un objet peut être **une fenêtre** sur un écran ou **un bouton** à cliquer ou **un personnage** dans un jeu vidéo etc. L'informaticien américain **Alan Key** travaille sur le langage *Smalltalk*. Dans ce langage de programmation, "tout est objet" : les chaines de caractères, les entiers, la mémoire...

1991 *Le langage Python*

Le néerlandais **Guido van Rossum** crée le langage multi plateformes[2] et multi paradigmes Python. Il est de typage dynamique c'est à dire que le type (flottant, entier etc.) se fait automatiquement lors de l'affectation d'une valeur à une variable. Exemple si a = 3 alors a est de type int.

[1] **Paradigme :** En informatique , c'est une manière de penser les problèmes et d'écrire des programmes pour les résoudre. Un exemple est le **paradigme impératif** dans lequel le programme est une liste d'instructions à suivre jusqu'à obtenir une solution.

[2] **Multiplateforme :** peut s'exécuter sous différents systèmes d'exploitation : Windows, Unix, macOS etc.

1.2 Constructions élémentaires dans un programme

- **Une affectation**

C'est une instruction qui donne une valeur à une variable.

- **Une séquence d'instructions**

Une séquence d'instructions est un bloc d'instructions groupées et qui sont exécutées dans l'ordre où elles sont écrites. En Python, les séquences d'instructions correspondent à des blocs indentés. Dans d'autres langages on met le bloc entre accolades, entre crochets ou entre parenthèses. Il est cependant conseillé de conserver l'indentation pour faciliter la lecture et la compréhension du code.

- **Une instruction conditionnelle**

Si une condition est vraie alors elle fait exécuter un bloc d'instructions. Sinon, le bloc n'est pas exécuté.

- **Une boucle bornée**

Elle fait exécuter un bloc d'instructions un nombre de fois déterminé à l'avance.

- **Une boucle non bornée**

Elle fait exécuter un bloc d'instructions un nombre de fois non déterminé puisque ce nombre est lié à une condition de maintien dans la boucle.

- **Un appel de fonction**

L'appel d'une fonction fait exécuter les instructions qu'elle contient. Si la fonction est directement utilisable alors elle est "native". Sinon, on doit l'importer depuis une "bibliothèque de fonctions".

2 Diversité et unité des langages de programmation

2.1 Paradigmes

Citons cinq des paradigmes utilisés en programmation :

- **Impératif** : Les opérations sont décrites sous la forme de suites ordonnées d'instructions.
- **Fonctionnel** : Le langage est basé sur l'application des fonctions.
- **Orienté objet** : Les "objets" représentent des concepts. Exemples : une fenêtre, un bouton.
- **Logique** : Le programme est structuré par la logique booléenne.
- **Évènementiel** : Le langage est axé sur la réaction à des événements survenant. Exemple : un clic sur un bouton.

2.2 Interprétation ou compilation

- Certains langages de programmation sont **interprétés** comme Python. Cela signifie qu'un *programme interpréteur* lit le code et effectue les opérations qui sont décrites, sans nécessairement le transformer en langage machine.
- D'autres langages comme le C++ doivent d'abord être **compilés**. Un *programme compilateur* doit d'abord transformer les lignes du **code source** écrit par un humain en **code objet.** Le compilateur optimise le code source en un code plus efficace. Le code objet est un ensemble d'instructions de bas niveau directement exécutables par un processeur spécifique. Cependant les fichiers objets obtenus doivent encore être liés par un programme éditeur de liens à d'autres bibliothèques tierces comme des bibliothèques C++ pour former un **code exécutable**.

2.3 Différences de syntaxe

- D'un langage à l'autre, la syntaxe[3] varie.

Exemple

Comparaison entre Python et JavaScript

Python	***JavaScript***

Fonction qui recherche le maximum
`def maximum(a, b):`
`  if a > b:`
`  return a`
`  else:`
`  return b`
	Fonction qui recherche le maximum
	`function maximum(a, b) {`
	`  if(a > b) {`
	`  return a`
	`  } else {`
	`  return b`
	`  }`
	`}`

- D'un langage à l'autre, il n'y a pas toujours les mêmes fonctions directement disponibles.

Exemple

Comparaison entre Python et JavaScript

Python	***JavaScript***

Fonction qui convertit en heures et minutes. Le quotient dans la division entière s'obtient directement par //
`def conversion(duree_en_minute):`
`  heures = duree_en_minute // 60`
`  minutes = duree_en_minute % 60`
`  return heures, minutes`
Exemple :
Si `duree_en_minute` vaut 125 alors
`duree_en_minute // 60` vaut 2
	Fonction qui convertit en heures et minutes. Le quotient dans la division entière s'obtient en prenant la partie entière du quotient décimal.
	`function conversion(duree_en_minute) {`
	`  let heures = Math.floor(duree_en_minute / 60);`
	`  let minutes = duree_en_minute % 60;`
	`  return [heures, minutes];`
	`}`
	Exemple :
	Si `duree_en_minute` vaut 125 alors
	`duree_en_minute / 60` vaut 2,08333...
	et `Math.floor(duree_en_minute / 60)` vaut 2
	Remarque : En JavaScript, le point virgule en fin de ligne n'est pas indispensable.

[3] **Syntaxe :** Ensemble des règles d'écriture d'un programme informatique permises dans un langage de programmation et formant la grammaire de ce langage.

3 Spécification d'une fonction

3.1 Prototype d'une fonction

Le **prototype** d'une fonction *est la déclaration* des éléments suivants :

- **Nom** de la fonction
- Liste des **arguments avec leur type**
- Liste des **valeurs renvoyées avec leur type**

L'utilisateur d'une fonction sait, en lisant le prototype, de quelle façon doit être appelée une fonction (nombre d'arguments et leur type). Si le langage compilé, le compilateur a besoin du prototype.

3.2 Documentation d'une fonction

La documentation des fonctions en Python s'appelle la *docstring.* On peut la lire en saisissant `help()`

Exemple : importation de la bibliothèque math et lecture de la documentation de la fonction sqrt

```
>>> import math
>>> help(math.sqrt)
Help on built-in function sqrt in module math:
sqrt(x, /)
    Return the square root of x.
```

On doit écrire une docstring au début de la fonction et **utiliser des noms de variables explicites**.

Exemple d'écriture d'une docstring

```
def conversion_duree(duree_en_minutes):
    """
    Prend en argument une durée en minutes et renvoie
    cette durée auformat heure - minutes.
    Exemple  : conversion_duree(125) renvoie (2, 5)

    Paramètre : duree_en_minte de type int
    -----------

    Renvoie : heures, minutes de type tuple
    ---------
    """

    heures = duree_en_minutes // 60
    minutes = duree_en_minutes % 60
    return heures, minutes
```

- Si on demande l'aide sur cette fonction dans la console :

```
>>> help(conversion_duree)
```

```
Help on function conversion_duree in module builtins:
conversion_duree(duree_en_minutes)
    Prend en argument une durée en minutes et renvoie
    cette durée auformat heure - minutes.
    Exemple  : conversion_duree(125) renvoie (2, 5)

    Paramètre : duree_en_minte de type int
    -----------
    Renvoie : heures, minutes de type tuple
    ---------
```

3.3 Les assertions

le mot assertion en français est synonyme d'affirmation. En programmation, c'est une condition qu'il est possible de vérifier lors de l'exécution d'une fonction. Le but est de tester si tout se passe comme prévu. On distingue les **préconditions** et les **postconditions**.

Préconditions

Les conditions **que doivent vérifier les arguments** pour qu'une fonction puisse être exécutée valablement.

Exemple :

```
def inverse(x):
    """
    Prend en argument un réel non nul et renvoie son inverse

    Paramètre : x de type float
    -----------

    Renvoie : y de type float
    ---------

    """
    assert x != 0, "l'argument doit être non nul."
    assert isinstance(x, float), "x doit être un flottant"
    y = 1/x
    return y
```

Expression booléenne qui doit être True

Message si l'expression booléenne est False

Expression qui est True lorsque x est un flottant

Message si l'expression booléenne est False

Dans la console :

```
>>> inverse(4)
0.25
>>> inverse(-2)
-0.5
>>> inverse(0)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<console>", line 19, in inverse
AssertionError: l'argument doit être non nul.
```

- Lorsque la fonction est mal utilisée (on demande ici de calculer l'inverse de zéro) l'instruction `assert` provoque **une erreur d'assertion** et stoppe le programme. **Le message prévu par le programmeur** s'affiche en rouge.
- Pendant la mise au point, le programmeur prévoit le plus de cas possibles de mauvaise utilisation de sa fonction ainsi que les messages d'erreur qui lui signaleront quelle assertion n'est pas vérifiée.

Postconditions

Les conditions **vérifiées par des résultats corrects.**

Exemple :

```
def essai(a, b):
    s = a + b
    assert 12 <= s and s <=30, "s doit être sur [12 ; 30]."
    return s
```

Dans cet exemple, un résultat correct de s doit être sur l'intervalle [12 ; 30].
Comme la condition $12 \leq s \leq 30$ porte sur le résultat de la fonction juste avant qu'il ne soit renvoyé, c'est une post condition.

4 Mise au point de programmes

Tester ses programmes est nécessaire pour essayer de trouver des "bugs" (ou "bogues").

- Un programme composé de plusieurs fonctions peut être testé dans sa totalité. Il s'agit alors d'un **test système**.
- Mais auparavant, on teste une fonction à la fois. Il s'agit des **tests unitaires**.
- Il est possible aussi de tester le programme dans sa totalité avec cependant qu'une partie des fonctions. Il s'agit de **tests d'intégration**.

- Les tests peuvent être soit exécutés manuellement par un humain, soit exécutés par des **programmes de tests** qui utilisent des bibliothèques.
- Les tests de fonctions doivent essayer de couvrir toutes les configurations possibles. Par exemple les instructions conditionnelles `if elif else` ont plusieurs branches qu'il faut toutes tester.
- Le test d'une fonction (appelé test unitaire) doit être relativement simple à imaginer. Sinon, il faudra décomposer la fonction en plusieurs fonctions plus simples.
- Enfin, en cas de réécriture de la fonction en simplifiant des blocs de code, la nouvelle fonction doit réussir les mêmes tests que l'ancienne. Ainsi on vérifie que l'on n'a pas régressé lorsqu'on a simplifié le code.

L'écriture d'une fonction doit toujours s'accompagner de l'écriture d'une fonction de test de cette fonction. Cependant, la réussite au test ne permet pas de prouver que la fonction est correcte dans tous les cas. La fonction de test a seulement pour but de vérifier les cas particuliers ou difficiles.

5 Utilisation de bibliothèques

- Une bibliothèque de fonctions est un ensemble de fonctions. Si une fonction est utile dans un programme alors on importe la bibliothèque qui la contient.

Exemple 1

En Python, les bibliothèques sont nommées "modules". Le module math contient la fonction sqrt.

```
import math
def f(x):
    return 2 * math.sqrt(x)
```

Exemple 2

On peut aussi importer toutes les fonctions avec la syntaxe `from math import *`
Dans ce cas **les fonctions sont appelées par leur nom sans préfixe**. Cependant cette forme est à éviter si on importe plusieurs modules, car *deux fonctions différentes* situées dans ces deux modules différents peuvent porter le *même nom* d'où une confusion possible.

```
from math import *
def g(x):
    return 3 * sqrt(x)
```

Exemple 3

Il est également possible d'importer un module en le renommant :

```
import numpy as np
```

(Le module "Numerical Python extensions" ou "NumPy" est une extension de Python créée pour étendre les possibilités de calcul de Python à certains types de tableaux).

Dès lors, le préfixe des fonctions du module numpy sera np. On dit que np est un **alias** de numpy.

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
```

Exemple 4

Enfin, on peut importer une seule fonction d'une bibliothèque particulière :

```
from math import sqrt
def h(x):
    return 4 * sqrt(x)
```

6 Exemple avec une bibliothèque et des assertions : Tri d'un tableau selon une colonne par la fonction sorted de Python

Soit le tableau

3	87	5	20	60
9	52	19	57	6
2	35	42	39	41
4	69	7	19	18

En Python, ce tableau peut être représenté par une liste de listes :

```
mon_tableau = [[3, 87, 5, 20, 60], [9, 52, 19, 57, 6], [2, 35, 42, 39, 41], [4, 69, 7, 19, 18]]

for ligne in mon_tableau:
    print(ligne)
```

affiche :

```
[3, 87, 5, 20, 60]
[9, 52, 19, 57, 6]
[2, 35, 42, 39, 41]
[4, 69, 7, 19, 18]
```

On écrit une fonction qui trie le tableau selon le numéro de colonne passé en paramètre.

Rappel : les numéros de colonnes sont des entiers allant de 0 à longueur d'une ligne - 1.

```
import operator  #  Bibliothèque contenant la fonction itemgetter
                 #  qui sert désigner la colonne selon laquelle trier.

def tri(tableau, colonne):

    """
    Cette fonction trie le tableau selon une des colonnes.

    Paramètres :
    -----------
        tableau : de type liste de listes d'entiers.
                C'est le tableau à trier.

        colonne : de type entier.
                C'est l'index de la colonne selon laquelle sera trié le
tableau

    Renvoie :
    --------
        tableau_trie : de type liste de listes d'entiers.
                    C'est le tableau trié.

    """

    assert isinstance(colonne, int), 'le numéro de colonne doit être entier.'
    assert 0 <= colonne <= len(tableau[0]) - 1, 'Numéro de colonne incorrect.'

    tableau_trie = sorted(tableau ,key=operator.itemgetter(colonne))
    return tableau_trie


mon_tableau = [[3, 87, 5, 20, 60], [9, 52, 19, 57, 6], [2, 35, 42, 39, 41], [4, 69, 7, 19, 18]]
```

Exemple : Trier le tableau selon la colonne d'index 3

```
mon_tableau_trie = tri(mon_tableau, 3)
for ligne in mon_tableau_trie:
    print(ligne)
```

affiche :

```
[4, 69, 7, 19, 18]
[3, 87, 5, 20, 60]
[2, 35, 42, 39, 41]
[9, 52, 19, 57, 6]
```