NSI – 1ere	COURS Séquence 1-BC : Spécification, mise au point de programmes	LFV

I : Traits communs et particuliers entre langages de programmation

Dans la séquence 1-A nous avons revu les six constructions élémentaires communes à tous les langages de programmation : séquences, affectation, conditionnelles, boucles bornées, boucles non bornées et fonctions.

Avant de poursuivre il faut que vous ayez conscience de l'existence de points communs entre tous les langages de programmation. Voici ci-dessous des exemples de syntaxe de ces constructions élémentaires dans trois langages : en python, en javascript et en C++. Bien entendu vous n'avez pas à apprendre la syntaxe pour javascript et C++. En revanche vous devez avoir vu et compris que les langages présentent de nombreux points communs.

python	java	C++
`reponse = 42`	`int reponse;` `reponse = 42;`	`int reponse;` `reponse = 42;`

`if condition :`
`  instruction A`
`  instruction B`
`else :`
`  instruction C`
`  instruction D`
`instruction E`
	`if ( condition ) {`
	`  instruction A;`
	`  instruction B;`
	`} else {`
	`  instruction C;`
	`  instruction D;`
	`}`
	`instruction E;`
		`if ( condition ) {`
		`  instruction A;`
		`  instruction B;`
		`} else {`
		`  instruction C;`
		`  instruction D;`
		`}`
		`instruction E;`

`for i in range(10) :`
`  instruction A`
`  instruction B`
`instruction C`
	`for ( int i=0; i<10; i=i+1 ){`
	`  instruction A;`
	`  instruction B;`
	`}`
	`instruction C;`
		`for ( int i=0; i<10; i=i+1 ){`
		`  instruction A;`
		`  instruction B;`
		`}`
		`instruction C;`

`while condition :`
`  instruction A;`
`  instruction B;`
`instruction C;`
	`while ( condition ){`
	`  instruction A;`
	`  instruction B;`
	`}`
	`instruction C;`
		`while ( condition ){`
		`  instruction A;`
		`  instruction B;`
		`}`
		`instruction C;`

`def chemin(a):`
`  ...`
`  chem = ch + ".jpg"`
`  return chem`
	`String chemin(int a){`
	`  String chem;`
	`  ...`
	`  chem = ch + ".jpg";`
	`  return chem;`
	`}`
		`string chemin(int a){`
		`  string chem;`
		`  ...`
		`  chem = ch + ".jpg";`
		`  return chem;`
		`}`

On voit donc un grand nombre de ressemblances entre ces trois langages. On remarque aussi des petites différences. Par exemple les blocs d'instructions sont délimités en Java et C++ par des accolades `{ … }` alors qu'en python ils sont délimités par un deux-points `:` et l'indentation du bloc.

Il y a une autre différence – beaucoup plus conséquente – entre python d'un côté et Java, C++ de l'autre. En effet on observe qu'en Java et C++ il faut déclarer le type des variables lors de leur définition. Ainsi dans le premier exemple, on déclare explicitement que la variable `reponse` est un entier.

Cela est également visible dans les définitions des fonctions. Ainsi lorsqu'on lit l'en-tête d'une fonction en Java ou en C++, on a tout de suite connaissance :

- du type des paramètres (ici le paramètre `a` est un entier)
- du type de la valeur de retour (ici une chaîne de caractères)

Si on regarde l'équivalent en python, la définition de la fonction `chemin` est moins claire. Ainsi au vu de l'en-tête de la fonction `chemin` en python, on est incapable de savoir quel est le type attendu du paramètre `a` et on est incapable de savoir si une valeur est retournée (il faut aller voir plus loins dans le code pour en avoir une idée).

Dans beaucoup de langages de programmation (mais pas en python), les *en-têtes* des fonctions doivent préciser :
- les *types* des paramètres,
- le *type* de l'éventuelle valeur de retour.

Que ce soit ou pas dans l'en-tête de la fonction, *prototyper* une fonction désigne le fait de décrire le type des paramètres attendus en entrée et le type de la valeur de retour en sortie*.

Le fait de prototyper une fonction permet aux autres programmeurs de voir rapidement quel est le type des entrées et de la sortie d'une fonction.
On notera par ailleurs qu'un prototype est parfois nécessaire dans certains langages (pas en python), il est dans ce cas utilisé lors du processus de mise en œuvre du programme par la machine.

* : lorsque la fonction ne fournit pas de valeur de retour, on peut utiliser selon les langages des "pseudo-objets" tels que `void` ou `None`

II Spécification d'une fonction, préconditions, postconditions

1) Spécification d'une fonction

Lorsqu'on programme une fonction, on se doit de préciser exactement ce que l'on attend d'elle. On dit que l'on doit préciser la *spécification* de cette fonction (cela renvoie à la notion de cahier des charges). Une *spécification* est élaborée en fonction du projet d'utilisation de la fonction. Une fois la *spécification* élaborée on peut commencer à penser à la façon de programmer l'intérieur de la fonction. Concrètement une spécification doit préciser :

- Le type des paramètres attendus en entrée ainsi que le type de la valeur de retour (c'est-à-dire, si vous avez suivi, le prototype de la fonction),

- D'éventuelles conditions portant sur les arguments qui seront donnés lors de l'appel de fonction (on parle de préconditions qui sont donc facultatives selon les fonctions),

- Ce que fait la fonction, exprimé en langage naturel (ce que l'on attend de la fonction). Cela se traduit par des propriétés qui doivent être vérifiées par la valeur de retour (on parle de postconditions).

2) Exemples de spécifications issues des documentations de référence

Exemple 1 : langage Python : méthode `time` du module `time`

`time`( )
Renvoie le temps en secondes depuis epoch sous forme de nombre à virgule flottante. La date spécifique de *epoch* et le traitement des secondes intercalaires (leap seconds) dépendent de la plate-forme. Sous Windows et la plupart des systèmes Unix, *epoch* est le 1er janvier 1970, 00:00:00 (UTC) et les secondes intercalaires ne sont pas comptées dans le temps en secondes depuis *epoch*. Ceci est communément appelé Heure Unix. Pour savoir quelle est *epoch* sur une plate-forme donnée, consultez gmtime(0).

Notez que même si l'heure est toujours renvoyée sous forme de nombre à virgule flottante, tous les systèmes ne fournissent pas l'heure avec une précision supérieure à 1 seconde. Bien que cette fonction renvoie normalement des valeurs croissantes, elle peut renvoyer une valeur inférieure à celle d'un appel précédent si l'horloge système a été réglée entre les deux appels.

- prototype :

- préconditions éventuelles :

- postconditions :

Exemple 2 : langage Javascript : méthode `floor` de l'objet `Math`

Remarque : en Javascript il n'y a pas de type `int` ou `float`. Il n'y a que le type `number`.

`floor(x)`

Parameters : x number.

Return value : A number representing the largest integer less than or equal to the specified number.

Note: `floor(null)` returns 0, not a `NaN`.

- prototype :

- préconditions éventuelles :

- postconditions :

Exemple 3 : langage C++ : méthode `sqrt`

Remarque : en C++, il existe 3 types de flottants de précision différente : `float, double, long double`

function `sqrt`

```
double sqrt (double x);
float sqrt (float x);
long double sqrt (long double x);
```

Parameters

`x` : Value whose square root is computed. If the argument is negative, a *domain error* occurs.

Return Value

Square root of `x`. If *x* is negative, a *domain error* occurs.

- prototype :

- préconditions éventuelles :

- postconditions :

Exemple 4 : langage python : méthode `choice` du module `random`

Remarque : en python les séquences sont par exemple les objets de type `str` ou `list`. Ils comportent un ou plusieurs éléments (pour rappel, les `str` comportent ainsi un ou plusieurs caractères …)

> `choice`(*seq*)
> Return a random element from the non-empty sequence *seq*. If *seq* is empty, raises IndexError

- prototype :

- préconditions éventuelles :

- postconditions :

3) Exercices de cours

Exercice 1

En comparant les deux documentations de référence ci-dessous concernant les méthodes `uniform` et `randint` du module `random` (langage python), déterminer tout ce qui distingue ces deux méthodes.

> `uniform`(*a*, *b*)
>
> Return a random floating point number *N* such that `a <= N <= b` for `a <= b` and `b <= N <= a` for `b < a`.
>
> The end-point value `b` may or may not be included in the range depending on floating-point rounding in the equation `a + (b-a) * random()`.

> `randint`(*a*, *b*)
>
> Return a random integer *N* such that `a <= N <= b`. Alias for `randrange(a, b+1)`.

Exercice 2

On a besoin d'une fonction `mystere` prenant un paramètre `n` et dont les appels ci-contre doivent fournir les valeurs de retour indiquées :

Appel effectué	Valeur de retour
`mystere(3)`	`"333"`
`mystere(7)`	`"7777777"`
`mystere(13)`	`"1313131313131"`

Cette fonction semble devoir réaliser une tâche assez précise. Trouver cette tâche puis réaliser une spécification de cette fonction. On fera émerger au moins une précondition qui semble nécessaire.

Remarque : l'élaboration de la spécification est un acte créatif qui suppose des choix. Il y a donc plusieurs variantes acceptables, il ne faut pas que cela vous arrête.

Exercice 3

On a besoin d'une fonction `mystere` prenant deux paramètres c et s et dont les appels ci-contre doivent fournir les valeurs de retour indiquées.

Appel effectué	Valeur de retour
`mystere("o", "précondition")`	4
`mystere("o", "prototype")`	2
`mystere("n", "fonction")`	2
`mystere("m", "programmation")`	6
`mystere("g", "Turing")`	5

Cette fonction semble devoir réaliser une tâche assez précise. Trouver cette tâche puis réaliser une spécification de cette fonction. On fera émerger au moins une précondition qui semble nécessaire.

Exercice 4

Dans les deux exercices précédents, des exemples supplémentaires d'appels à la fonction `mystere` auraient dû être donnés afin d'indiquer plus clairement ce qu'il est censé se passer dans certains cas problématiques.

Lister de tels exemples pour chacun des deux exercices.

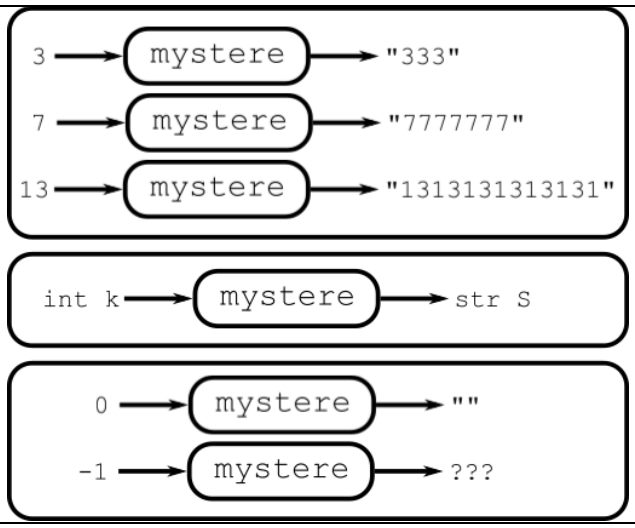
III Démarche de conception d'une fonction

Lorsqu'un programmeur a besoin d'une fonction, il doit réfléchir comme dans les exercices 2 et 3 ci-dessus :

- Il a ainsi en tête les valeurs de retour attendues sur des exemples "classiques et usuels" d'arguments passés lors de l'appel.
- Cela lui permet de prototyper sa fonction
- Et également de connaître les postconditions de sa fonction
- Puis il doit réfléchir aux exemples moins classiques qui pourraient survenir et à la façon dont ces cas – potentiellement problématiques – doivent être gérés.
- Cela lui permet d'envisager des préconditions
- Il a désormais établi la spécification de sa fonction.
- Il peut alors commencer à réfléchir à la programmation de sa fonction

1ere étape : exemples usuels

2nde étape : exemples usuels + prototype



3eme étape : exemples usuels + prototype + cas particuliers

Fonction `mystere`
`string mystere(int k)`
Paramètres : `k` est un entier positif ou nul
Valeur de retour : chaîne de caractères `S` comportant `k` caractères obtenus en répétant le motif des caractères du nombre `k`.
Pour k=0, renvoie une chaîne de caractères vides.
Pour k<0, renvoie une erreur.
Exemple :

```
>>> mystere(13)
"1313131313131"
```

4eme étape : on élabore la spécification : prototype, préconditions éventuelles et postconditions.

Quel algorithme vais-je utiliser ?
Quelles méthodes python vont m'être utiles ?
Quelles boucles ?
??
?

5eme étape : n peut commencer à réfléchir à la programmation une fois que l'on a **l'étape 3 ET l'étape 4** bien en tête.

Remarque : On aurait pu choisir d'accepter k<0 en renvoyant une chaîne de caractères vide. Ici c'est le programmeur qui doit choisir en fonction de son projet …

Une erreur de programmeur débutant est de vouloir programmer directement sans avoir formalisé la spécification de sa fonction. Généralement cela se traduit par des confusions (en particulier sur les types des variables utilisées) qui conduisent à l'impossibilité de formaliser le code à programmer …

IV Tests et assertions

En informatique, lors de la conception d'une application, il y a plusieurs catégories de tests :
- les test *unitaires* s'appliquant à une méthode ou fonction isolée du reste de l'application,
- les test d'*intégration* s'appliquant à une méthode ou fonction intégrée à l'application,
- les tests *fonctionnels* s'appliquant à une fonctionnalité complète et décrits par une succession d'actions utilisateur.

Nous ne parlerons ici que des tests *unitaires*.

Compte-tenu de la démarche de conception d'une fonction, il est souvent judicieux ***– avant même de commencer à coder une fonction*** – de coder quelques tests *unitaires* qui pourront être effectués à chaque nouvelle version de la

fonction. Pour cela il suffit de se baser que ce qui doit être fait à l'étape 3 de la conception (voir III) en prenant *suffisamment* d'exemples usuels et de cas particuliers.

Techniquement, on utilise fréquemment des *assertions* pour mettre en place des tests. Les *assertions* existent dans beaucoup de langages de programmation et permettent de renvoyer une erreur lorsqu'une condition n'est pas vérifiée.

Voici un exemple de tests qui pourraient être utilisés pour tester la fonction `mystere` évoquée précédemment :

```
assert(mystere(3)=='333')
assert(mystere(7)=='7777777')
assert(mystere(13)=='1313131313131')
assert(mystere(0)=='')
```

Si une des conditions mentionnées dans les *assertions* n'est pas vérifiée, alors l'*assertion* lève une *exception* (`AssertionError`).

ATTENTION : Une fonction peut très bien passer un jeu de tests sans pour autant être correcte. Les tests permettent normalement de valider le fonctionnement "usuel" mais ne peuvent pas forcément valider tous les cas qui sont susceptibles de se produire.

Exercice 5 :

Par rapport à la spécification de la fonction `mystere`, il y a un cas qui n'est pas testé. Lequel ?

Compléter dans l'encadré ci-dessous l'explication qui vous a été donnée quant à ce cas non testé :

La notion de …. n'est pas au programme. De plus, gérer les … dans les tests en python demanderait un petit travail technique qui compliquerait les choses. C'est pourquoi dans la suite du cours nous considérerons uniquement les cas où l'on teste des valeurs de retour. Nous ne parlerons pas de la gestion des ….

Pour finir ce cours, et avant de le mettre en pratique sur un TP, il faut impérativement mentionner que les assertions peuvent aussi servir à vérifier si des préconditions ou des postconditions sont bien vérifiée. Voici par exemple comment on peut lever une *exception* (`AssertionError`) lorsque l'argument passé à la fonction `mystere` est négatif :

```
def mystere(k):
    ''' Retourne une chaîne...
    assert(k>=0)
    …
    …
    …
```

Remarque 2:

Les assertions sont réservées aux versions des applications en développement (c'est-à-dire en cours de fabrication). Une fois que l'application est mise en production (c'est-à-dire mise en service auprès des utilisateurs) les assertions ne doivent plus être utilisées. Pour cette raison il existe souvent des bibliothèques (modules `pytest` *et* `unittest` *en python) qui permettent de gérer efficacement les assertions. Par exemple en offrant la possibilité de les désactiver facilement lors de l'exécution du programme.*
Cette année nous n'utiliserons pas de telles bibliothèques, mais il est important d'avoir conscience de leur existence.

Remarque 2 :

Si vous codez une fonction sans réussir à mettre des tests unitairtes en place, c'est sans doute que votre fonction est trop complexe ou mal pensée …