NSI – 1ere	SYNTHESE DE COURS Séquence 1-A : Constructions élémentaires	LFV

Tous les langages de programmation partagent un meêm ensemble de constructions élémentaires.

Séquences d'instructions

Dans un programme informatiques les *instructions* sont écrites ligne après ligne et sont exécutées de haut en bas.

Affectations

Les *affectations* permettent d'affecter des valeurs à des *variables*. On distingue plusieurs *types* de variables. En ce début d'année, nous connaissons les *types entier, flottant, booléens et chaîne de caractères*. En python ce sont les types `int, float, bool` et `str` qui peuvent être obtenus grâce à la méthode `type()`.

```
a = 6
a = a + 7
b = a * 3
```

Conditionnelles

Les *conditionnelles* sont des instructions qui ne sont exécutées que si une certaine *condition* est vérifiée. Dans la majorité des langages de programmation une conditionnelle est programmée grâce au mot clef `if` .
En python on dispose de la syntaxe ci-contre où les deux `elif` et le `else` sont facultatifs.

```
if condition1 :
      instruction A
elif condition2 :
      instruction B
elif condition3 :
      instruction C
else :
      instruction D
```

Boucles bornées

Les *boucles bornées* permettent de répéter des instructions. Dans la majorité des langages de programmation une boucle bornée est programmée grâce au mot clef `for`.

```
for x in iterable:
      instruction A
      instruction B
```

En python on dispose de la syntaxe ci-contre où *iterable* est un ensemble pouvant être parcouru (par exemple `range(a, b)` ou chaîne de caractères). À l'intérieur de la boucle on peut si besoin utiliser la valeur de `x` qui parcourt `iterable` et donc varie automatiquement à chaque nouveau tour de boucle.

Boucles non bornées

Les *boucles non bornées* permettent de répéter des instructions tant qu'une condition est vérifiée. Dans la majorité des langages de programmation une boucle non bornée est programmée grâce au mot clef `while`.

```
while condition :
      instruction A
      instruction B
```

En python on dispose de la syntaxe ci-contre où la boucle est répétée tant que la condition est vraie.
Il faut faire attention aux boucles non bornées car on peut facilement obtenir une boucle infinie non désirée.

Fonctions

Les fonctions sont des blocs d'instructions qui peuvent être *appelés* ailleurs dans un programme. Elles acceptent souvent un ou plusieurs *paramètres en entrée* et peuvent *retourner une valeur en sortie*.

```
def somme(a, b, c):
      '''Retourne la somme des 3 entiers a, b, c'''
      s = a + b + c
      return s
```

En python, dans *l'en-tête de leur définition*, on utilise le mot clef `def` et on indique les noms des paramètres utilisés entre parenthèses.
En python, lors de l'*appel* d'une fonction, on précise entre parenthèses les valeurs des *arguments qui remplacent les paramètres* lors de l'exécution.

```
>>> somme(5, 3, 4)
12
```

En python, l'éventuelle *valeur de retour* est précisée grâce au mot clef `return`. Ce mot clef est particulier car lorsque l'instruction comportant `return` est rencontrée, l'exécution de la fonction est arrêtée.
En python, on précise *sous l'en-tête* de la fonction ce qu'elle fait grâce à une *docstring* encadrée par des *triple quotes*.

Autres points abordés

Itérable `range` :

`range(a=0, b [,p])` correspond aux entiers compris entre `a` (compris) et `b` (non compris) avec un pas `p`.

- `range(20)` : représente les entiers de 0 à 19 (0 et 19 compris)
- `range(5, 20)` : représente les entiers de 5 à 19 (5 et 19 compris)
- `range(5, 20, 3)` : représente les entiers de 5 à 17 avec un pas de 3 (5 et 17 compris)

Module `random` :

Le module `random` permet de générer des nombres aléatoires. Par exemple, la méthode `randint(a, b)` permet de générer un entier alétoire compris entre `a` (compris) et `b` (compris).

```
import random

resultat = random.randint(1, 10)
```

Opérateurs de comparaison :

```
a<b
a>b
a<=b
a>=b
a==b
a!=b
```

Opérateur d'appartenance `in` :

```
>>>"k" in "cookie"
True
>>>"k" in "mangoose"
False
>>>19 in range(10, 100, 9)
True
```

Quotient `//` et modulo `%` :

```
>>>67//10
6
>>>67%10
7
>>>for i in range(100):
       print(i%3)
0
1
2
0
1
2
0
1
...
```

Ajouter (on dit "concaténer") deux chaînes de caractères entre elles :

```
>>>"ABCDE"+"F"
ABCDEF
```

Module `string` (pas à connaître car pas souvent utile) :

```
>>>import string
>>>string.ascii_lowercase.index("e")
4
>>>string.ascii_lowercase[10]
'k'
```