NSI – Révisions 1 – Programmation de base

Question 01 Donner la signification et des exemples d'utilisation des types natifs de base **`int`**, **`float`**, **`bool`** et **`str`**.

Question 02 Donner la signification et des exemples d'utilisation des types natifs structurés **`list`**, **`tuple`** et **`dict`**.

Question 03 Expliquer la signification des opérateurs suivant : **`==`** **`!=`** **`=`** **`is`** **`is not`**

Question 04 Pour chacune des variables **n**, **rep**, **k** et **x**, dire s'il s'agit d'une variable globale ou locale.

```
def calcul(n):
    rep = 0
    for k in range(1, n) :
        rep = rep + k
    return rep*10

x = calcul(5)
```

Question 04bis Expliquer la valeur que prend **x** à la fin de l'exécution du programme.

Question 05 Quelqu'un modifie le nom de **k** en **x**. Expliquer si cela va causer une erreur ou pas.

```
def calcul(n) :
    rep = 0
    for x in range(1, n) :
        rep = rep + x
    return rep*10

x = calcul(5)
```

Question 06 : Donner la table de vérité du **and** et du **or.** Puis du **nand, nor** et **xor.**

Question 07 : Quelle est la différence entre commentaires et documentation ?

Question 08

Réaliser la documentation longue de la fonction **`moyenne()`** dont voici le prototype :
`moyenne(t:list) → float`

Vous devrez notamment y placer un exemple de fonctionnement.

Voici cette documentation en mode documentation rapide :

```
def moyenne(t:list) -> float :
    """Renvoie la moyenne des nombres (int|float) placés dans le tableau t"""
    pass
```

Question 09 : réaliser la fonction ainsi qu'une fonction tester() qui utilise assert pour tester si la fonction répond correctement.

Question 10 : utiliser le module doctest pour automatiser les tests.
from doctest import testmod

```
import doctest
doctest.testmod()
```

ou

```
from doctest import testmod
testmod()
```

NSI - Révisions 2 - Algorithmique

Recherche du coût

Le **coût** (ou parle aussi de **complexité**) permet d'avoir une idée de la façon dont le nombre d'instructions varie lorsqu'on augmente le nombre de données à traiter par l'algorithme.

On cherche souvent d'abord à étudier **le coût dans le pire des cas** : on se place dans la configuration la plus défavorable.

Voici des exemples de coûts qui permettent de classer les performances des algorithmes :

Complexité	Nom du coût	Evolution des données	Evolution du temps d'exécution
$\Theta(\log_{10} n)$ ou $\Theta(\log n)$	Logarithmique (base 10)	1.10^3 vers 1.10^6 1.10^6 vers 1.10^9	Exécution x 2 uniquement Exécution x 1,5 uniquement
$\Theta(\log_{10} n)$ ou $\Theta(\log n)$	Logarithmique (base 2)	1.10^3 vers 1.10^6 1.10^6 vers 1.10^9	Exécution x 2 uniquement Exécution x 1,5 uniquement
$\Theta(n)$	Linéaire	1.10^3 vers 1.10^6 1.10^6 vers 1.10^9	Exécution x 1000 Exécution x 1000
$\Theta(n \log_X n)$	Quasi-linéaire	1.10^3 vers 1.10^6 1.10^6 vers 1.10^9	Exécution x 2000 Exécution x 1500
$\Theta(n^2)$	Quadratique	Fois 1000	Exécution x 1000^2, soit 1 million
$\Theta(n^X)$	Polynomial	Fois 1000	Exécution x 1000^X
$\Theta(x^n)$	Exponentiel	1 vers 1000	Exécution x5.10^{300} si on a 2^X
$\Theta(n !)$	Factoriel (10*9*8..*1)	1 vers 1000	Python ne parvient même pas à le calculer...

Equivalence entre POUR et TANT QUE

On peut écrire une **BOUCLE BORNEE** sous la forme d'un **POUR** ou d'un TANT QUE.

On peut écrire une **BOUCLE NON BORNEE** sous la forme d'un **TANT QUE** ou d'un POUR+return.

Variant et preuve de TERMINAISON

Prouver la terminaison d'un algorithme veut dire qu'on prouve qu'**il va s'arrêter** pour **n'importe quelle entrée respectant les préconditions**. Aucun risque de boucle infinie.

Pour prouver la terminaison, il faut vérifier que toutes les boucles peuvent s'écrire de cette façon :

`TANT VARIANT > 0` avec VARIANT sous forme **u_N suite strictement décroissante d'entiers**

Invariant et preuve de CORRECTION

INVARIANT (Vocabulaire) : une **propriété P** caractéristique des données du problème. L'invariant doit être vrai :

avant la première boucle ;
à la fin de chaque tour de boucle ;
après avoir effectué tous les tours de boucle.

La démonstration se fait en trois temps :

INITIALISATION : On montre que **P_0 est vraie**.
Qu'est-ce que **P_0** ? Le fait que l'invariant soit vrai avant de rentrer dans la boucle (0 tour effectué).

CONSERVATION (hérédité) : on suppose **P_k** vraie et on montre que **P_k vraie → P_{k+1} vraie**.
Cela veut dire qu'on fait l'hypothèse que P est vraie après k tours de boucle. Il faut montrer que P reste vraie lorsqu'on effectue un tour de boucle en plus.

TERMINAISON (conclusion) : on montre qu'après avoir fait les boucles, le problème est résolu.

NSI - Révisions 3 – Tri par insertion

Principe général : on ordonne un ensemble d'une carte, puis de 2 cartes, puis de 3 cartes...

→ DEBUT :
- → on prend la première carte pour former le sous-tableau trié.
- → les autres cartes forment le sous-tableau non-trié.

→ EXECUTION :
- → on enlève la carte suivante dans le sous-tableau non trié.
- → on déplace vers la droite les cartes déjà triées qui sont supérieures à notre nouvelle carte
- → on insère notre nouvelle carte dans la case disponible

Coûts :
→ PIRE DES CAS : Coût quadratique $\Theta(n^2)$ → MEILLEUR DES CAS : Coût linéaire $\Theta(n)$

Algorithme :

Vous devez être capable de

- fournir l'algorithme
- implémenter l'algorithme en Python, l'algorithme étant donné

Algorithme de tri par insertion, non commenté

```
POUR i variant de 1 à (longueur - 1)
    cle ← tableau[i]
    j ← i - 1
    TANT QUE j ≥ 0 et que tableau[j] > cle
        tableau[j+1] ← tableau[j]
        j ← j - 1
    Fin TANT QUE
    tableau[j+1] ← cle
Fin POUR
Renvoyer VIDE (∅)
```

Remarque sur la traduction des for dans le sens Algortihme vers Python

** en algo, la fin est incluse*
** en python, la fin est exclue*

```
Pour i variant de 1 à (longueur - 1)  -----→      for i in range(1, longueur)
```

NSI - Révisions 4 – Tri par sélection

Principe général : on cherche la plus petite, puis la 2e plus petite, puis la 3e plus petite...

→ DEBUT : le sous-tableau trié est vide.

→ EXECUTION : on cherche à remplir la prochaine case du sous-tableau trié. Pour cela :
- → on considère d'abord que la plus petite est bien la carte présente dans cette case.
- → on cherche la plus petite carte dans le sous-tableau non trié
- → on inverse les deux cartes si nécessaire en les plaçant dans la case de l'autre.

Côuts :
→ PIRE DES CAS : Coût quadratique $\Theta(n^2)$ → MEILLEUR DES CAS : Coût quadratique $\Theta(n^2)$

Algorithme :

Vous devez être capable de

- fournir l'algorithme
- implémenter l'algorithme en Python, l'algorithme est donné dans ce cas

Algortihme du tri par sélection

```
→ debut est l'index (flèche en vert clair dans l'animation)
POUR debut variant de 0 à (longueur - 1)
    index_min ← minimum(tableau, debut)
      ↳ on cherche l'index de la plus petite valeur présente
    intervertir(tableau, debut, index_min)
      ↳ on intervertit les valeurs situées aux index debut et index_min
  Fin POUR
Renvoyer VIDE (∅)
```

Algorithme de `minimum(tableau, debut)`

```
→ debut est l'index (flèche en vert clair dans l'animation)
index_min ← debut
POUR index variant de (debut + 1) à (longueur - 1)
    SI tableau[index] < tableau[index_min]
        index_min ← index
    Fin SI
  Fin POUR
Renvoyer index_min
```

Algorithme de `intervertir(tableau, debut, index_min)`

Algorithme

```
temp ← tableau[debut]
tableau[debut] ← tableau[index_min]
tableau[index_min] ← temp
Renvoyer Vide
```