1 Compréhension du cours

Vous devez répondre à la question suivante en réfléchissant, sans le support d'une exécution Python.

On souhaite écrire une fonction qui renvoie `True` ssi la liste d'entiers passée en paramètre contient au moins un élément positif ou nul.

1. Quels tests proposez-vous pour cette fonction (cas nominal · aux et/ou cas particulier · s) ?

Les fonctions suivantes (sans précondition) sont codées dans l'intention de réaliser la spécification ci-dessus.

2. Pour chaque fonction :

- indiquer si son code est conforme aux bonnes pratiques de l'UE
- en réfléchissant au code ou en déroulant à la main l'exécution sur les données de test, indiquer si le code est correct (= la fonction renvoie un résultat conforme à sa spécification pour tout paramètre satisfaisant sa précondition)
- si le code n'est pas correct : donner un ou plusieurs tests qui mettent en évidence le problème, ainsi qu'un ou plusieurs tests qui **ne mettent pas** en évidence le problème et peuvent laisser penser que la fonction est correctement codée.

```
def foo1(liste:list[int]) -> bool:
    for elem in liste:
      if elem >= 0:
         return True
      else:
         return False

def foo2(liste:list[int]) -> bool:
    for elem in liste:
      if elem >= 0:
         res = True
      else:
         res = False
    return res
```

```
def foo3(liste:list[int]) -> bool:
   i = 0
   while i < len(liste):
      if liste[i] >= 0:
         return True
      i = i + 1
   return False

def foo4(liste:list[int]) -> bool:
   i = 0
   while liste[i] < 0 and i < len(liste):
      i = i + 1
   return i < len(liste)
```

3. Que pensez-vous des fonctions suivantes ?

```
def foo5(x : str, u : str) -> list[str]:
   t = []
   for i in x:
      if i != u:
         t.append(i)
   return t
```

```
def foo6(a:list[int]) -> bool:
   x = 0
   while x < len(a) and a[x] != -1:
      x = x + 1
   return x < len(a)
```

2 Exercices de programmation

Certains des exercices suivants n'impliquent pas du tout d'itérable. D'autres se rapportent aux schémas algorithmiques vus en amphi pour interrompre le parcours séquentiel d'un itérable. D'autres enfin impliquent un parcours d'itérable, mais plus complexe qu'un simple parcours séquentiel. Vous devez réfléchir à ce point.

Par ailleurs utiliser l'approche classique : prévoir des tests et dérouler des exemples à la main avant d'écrire le code (ça peut aussi vous aider à réfléchir au point précédent).

2.1 Primalité

Un entier p est dit premier si il est supérieur ou égal à 2 et si aucun des entiers compris entre 2 et $p-1$ inclus ne divise p.

4. Écrire un prédicat `est_premier_naif` qui prend en paramètre un entier `p` positif et renvoie `True` ssi `p` est premier.

Les intervalles sont des itérables indexables, par exemple `range(2, 6)[1]` vaut `3`.

5. Écrire un prédicat `au_moins_un_premier` qui prend en paramètre un intervalle et renvoie `True` ssi l'intervalle contient au moins un nombre premier. Par exemple l'intervalle, `range(12)` contient 2, qui est premier, alors qu'aucun des trois éléments de l'intervalle `range(8, 11)` (qui sont 8, 9 et 10) n'est premier.

2.2 Autour des chaînes

6. Écrire une fonction `genere_binaire` qui prend en paramètre un entier `n` positif et construit une chaîne de taille au moins $2n$ en tirant aléatoirement le prochain élément dans l'ensemble $\{0, 1\}$ jusqu'à ce que la liste contienne au moins `n` occurrences de `'0'` et `n` occurrences de `'1'`. Par exemple on pourra obtenir : `"01110"` pour n valant 2, mais pas `"011101"`.

7. Écrire un prédicat `apparait` qui prend en paramètre 2 chaînes de caractères `mot` (non vide) et `texte` et renvoie `True` ssi `mot` apparaît dans `texte` pourvu qu'on en supprime des caractères. Par exemple, `'chat'` apparaît dans `'cheminant'` mais n'apparaît pas dans `'chemin'`.

2.3 Autour de listes d'entiers

8. Écrire une fonction `prefixe_somme` qui prend en paramètre une liste d'entiers positifs `liste` et un entier positif `n` et renvoie le plus petit préfixe de `liste` tq la somme de ses elts est supérieure ou égale à `n`. Par exemple :

```
>>> prefixe_somme([1, 2, 3], 1)
[1]
>>> prefixe_somme([1, 2, 3], 4)
[1, 2, 3]
>>> prefixe_somme([1, 2, 3], 10)
[1, 2, 3]
>>> prefixe_somme([1, 2, 3], 0)
[]
>>> prefixe_somme([], 10)
[]
```

9. Écrire une fonction `somme_vaut_n` qui prend en paramètre une liste d'entiers strictement positifs et un entier `n` et renvoie `True` ssi la somme des éléments de la liste vaut `n`. L'idée n'est pas de calculer la somme des éléments de toute la liste puis de comparer le résultat avec `n`, mais d'interrompre le parcours dès qu'on est sûr que la somme ne peut pas valoir `n`. Par exemple :

```
>>> somme_vaut_n([1, 2, 3], 6)
True
>>> somme_vaut_n([1, 5, 2], 3)
False
>>> somme_vaut_n([1, 1, 1], 6)
False
>>> somme_vaut_n([1, 5, 1], 6)
False
>>> somme_vaut_n([], 0)
True
>>> somme_vaut_n([], 2)
False
```

Python permet de comparer des listes d'entiers avec l'opérateur `<` :

```
>>> [2, 5, 4, 6, 7] < [5]
True
>>> [5, 5, 2] < [5, 5, 7]
True
>>> [5, 5, 2] < [5, 5, 1]
False
>>> [5, 6] < [5, 6, 7]
True
```

L'algorithme est le suivant :

- les entiers à l'indice 0 sont comparés : si celui de gauche est strictement inférieur à celui de droite, la liste de gauche est strictement inférieure à celle de droite ;
- en cas d'égalité des entiers à l'indice 0, on renouvelle l'opération avec les entiers à l'indice 1, *etc* ;
- si l'une des listes est un préfixe (strict) de l'autre, la moins longue est strictement inférieure à la plus longue, en cas d'égalité aucune n'est strictement inférieure à l'autre.

10. Écrire un prédicat `est_inferieure_strict` qui prend en paramètre 2 listes d'entiers `liste1` et `liste2` et renvoie `True` ssi `liste1 < liste2`.