IUT Nantes
Pôle Sciences et technologie

Nantes Université

BUT Info 1

Exercices Mathématiques discrètes

2023-2024

Travaux Pratiques

1 Découverte de Python

Exercice 1

L'objectif de ce TP est de vous familiariser avec Python.

Python peut être lancé de deux manières :

- via la console : tapez `python3` dans un terminal
- en exécutant un fichier : tapez `python3 nom_fichier.py` dans un terminal

1.1. Tapez les commandes suivantes :

```
>>> 5+7
>>> 34/9
>>> min(3,5)
>>> sqrt(9)
```

Les fonctions mathématiques ne sont pas chargées par défaut : il faut importer le module `math` :

1.2. Tapez les commandes suivantes :

```
>>> import math
>>> math.sqrt(9)
```

Pour utiliser la fonction `sqrt` du package `math`, il y a 3 possibilités :

- importer le package `math` puis utiliser la commande `math.sqrt`
- importer le package `math` en lui donnant un alias :

  ```
  >>> import math as mt
  ```

  puis utiliser la commande `mt.sqrt`. Ceci permet d'éviter des noms trop long de package.
- importer directement certaines ou toutes les fonctions du package `math` :

  ```
  >>> from math import *
  ```

  puis les utiliser directement sans préfixe : `sqrt`. ATTENTION !! : ceci est risqué si vous importez plusieurs packages avec des fonctions portant le même nom.

1.3. Calculer les valeurs suivantes :

- $\sqrt{5}$
- 2^6
- $17 \ mod \ 4$

Les variables s'affectent par le signe `=` :

```
>>> a = 3+4
```

L'affichage d'une variable dans la console se fait juste en tapant son nom ou `print(nom_variable)`

```
>>> a
>>> print(a)
```

En Python, les deux valeurs binaires sont `True` et `False`.

1.4. (Opérateurs binaires) Exécutez les commandes suivantes :

- `True and False`
- `True or False`
- `not(True) or False and True`

==	teste l'égalité entre deux valeurs
!=	teste la différence entre deux valeurs
<, <=, >, >=	teste les relations d'ordre entre deux valeurs

Comparaisons

Exercice 2 `if`, `while`, `for`, `range` et fonctions

Voici un exemple d'utilisation en python de l'instruction `if` :

```
if x>0 :
        y=x
        print("Cas positif")
elif x<0 :
        y=-x
        print("Cas négatif")
else :
        y=0
        print("Cas nul")
print("Fin")
```

Noter l'absence d'accolades et l'utilisation de l'indentation : les blocs de l'instruction conditionnelle ne sont distinguables que par l'indentation. Il en est de même pour les boucles `for`, `while` et pour l'écriture des fonctions.

Voici un exemple de fonction écrite en python :

```
def f(x) :
        if x >= 0 :
                return x
        else :
                return -x
```

2.1. Dans un fichier `tutoriel.py`, écrivez une fonction g permettant de calculer :

$$g(x) = \begin{cases} \sqrt{-x} & \text{si } x < 0 \\ \sqrt{x} & \text{sinon} \end{cases}$$

Python est surtout un langage de script. Si vous souhaitez tester votre fonction, il vous suffit de rajouter, par exemple,

```
print(g(-4))
```

à la fin de votre fichier et de le lancer dans le terminal par la commande :

```
$ python3 tutoriel.py
```

La commande `for` s'utilise de la manière suivante :

```
for i in [4,8,-3] :
        print(i)
```

2.2. Afficher ce que donne les commandes :

- ```
  for i in range(6) :
      print(i)
  ```
- ```
  for i in range(5,10) :
      print(i)
  ```
- ```
  for i in range(30,3,-7) :
      print(i)
  ```

2.3. Écrivez une fonction h prenant en paramètre un entier positif x et affichant tous les entiers multiples de 7 compris entre 0 et x.

Exercice 3 Tuples et listes

En python, une liste s'écrit entre crochets, avec ses valeurs séparées par des virgules :

```
>>> squares = [1, 4, 9, 16, 25]
```

3.1. Déterminer ce qu'affichent et modifient les commandes suivantes :

- squares[0]
- squares[1:3]
- squares[:2]
- squares[2:]
- squares[−1]
- squares+36
- squares∗2
- squares∗3.5
- squares+[36]
- squares.append(36)
- squares.append([49])
- squares.extend([64])
- squares.remove(25)
- squares.pop(4)
```

3.2. Construire une fonction `compte` prenant en paramètres un élément `el` et une liste `liste` et retournant le nombre d'éléments de `liste` égaux à `el`.

3.3. Construire une fonction `carres` prenant en paramètre une liste `liste` et retournant une liste contenant les carrés des éléments de `liste`.

Les tuples (ou n-uplets) sont une deuxième structure classique en python, voici un exemple de déclaration :

```
>>> mon_tuple = (1,8,8,64,125)
```

Les sets (ou ensembles) sont une troisième structure en python, voici un exemple de déclaration (à remarquer notamment la façon dont est géré le 8 qui apparaît en doublon) :

```
>>>mon_set={1,8,8,64,125}
```

3.4. Déterminer quelles actions possibles sur les listes ne le sont pas sur les tuples et/ou sur les ensembles.

2 Logique

Exercice 4

4.1. En utilisant seulement des conditionnelles (`if`) et sans les opérateurs logiques de `python` (`and`, `or`, `not`), coder les opérateurs logiques suivants :

- `et(x,y)` qui renvoie `x` $\wedge$ `y`
- `ou(x,y)` qui renvoie `x` $\vee$ `y`
- `non(x)` qui renvoie $\neg$`x`
- `implique(x,y)` qui renvoie `x` $\Rightarrow$ `y`

4.2. A partir des fonctions définies précédemment (soit en utilisant une fonction composée à partir de celles-ci), retrouver la table de vérité étudiée en cours pour déterminer la satisfiabilité de la formule $F_1 \wedge F_2$ avec $F_1 = \neg p \Rightarrow (q \wedge r)$ et $F_2 = q \Rightarrow (\neg p \wedge \neg r)$. On générera, à partir des listes donnant les valeurs respectives de p, q et r sur l'ensemble des lignes de la table, cette table de deux manières :

- en réalisant une boucle sur l'ensemble des lignes et en ajoutant le résultat à une liste initialement initialisée comme vide
- en définissant une liste de résultat en compréhension à l'aide de l'itérateur `zip(p,q,r)` renvoyant toutes les valuations à étudier sur les différentes lignes :

```
resformule=[composeefonctions(x,y,z) for x,y,z in
zip(p,q,r)]
```

4.3. Afin d'envisager la généralisation du procédé précédent ($\rightarrow$ SAE) on souhaite définir une fonction permettant de générer la liste de toutes les valuations (une liste de listes) pour un nombre de variables quelconque passé en argument. On pourra par exemple pour cela envisager :

- d'alimenter une liste initialisée comme liste contenant une liste vide et de l'alimenter en faisant une boucle sur les différentes variables pour multiplier systématiquement par deux les éléments de la liste en considérant les deux possibilités de valeurs logiques pour cette variable (avec un procédé semblable on peut aussi procéder par récursivité)
- de créer directement les listes de valuations en exploitant l'écriture en binaire du numéro de chaque ligne, en utilisant par exemple la fonction `format`

3 Ensembles

Exercice 5 Parcours d'ensembles en `python`

Dans la question 5.1, on utilise des `set` pour représenter des ensembles.
Le but de ces TP est de vous faire manipuler les concepts de base de programmation, aussi il est important que vous codiez les fonctions sans utiliser les fonctions prévues dans `python` pour manipuler les `set`.

5.1. Écrire en `python` les fonctions suivantes :

a. `affiche_col(E)` : fonction qui affiche un par un les éléments de l'ensemble `E` passé en argument.

b. `membre(x,E)` : fonction qui prend un élément `x` et un ensemble `E` en paramètres et qui renvoie `True` si `x` est dans `E`, `Faux` sinon.

c. `somme(E)` : fonction qui calcule, quand c'est possible, la somme des éléments d'un ensemble `E` ;

d. `taille(E)` : calcule le nombre d'éléments d'un ensemble `E` ;

5.2. Un ensemble est un regroupement d'éléments sans doublon et sans ordre :

a. écrivez une fonction `est_ensemble(l)` prenant en paramètre une liste l et qui retourne `True` si la liste `l` ne possède pas de doublon, `False` sinon.

b. écrivez une fonction `suppr_doublon(l)` qui retourne l'ensemble issu de la liste `l`, c'est à dire sans doublon.

Exercice 6 Opérations ensemblistes en `python`

6.1. Écrire en `python` les fonctions booléennes suivantes :

a. `inclus(A,B)` : renvoie `True` si l'ensemble représenté par `A` est inclus dans celui représenté par `B`, `False` sinon.

b. `egal(A,B)` : renvoie `True` si l'ensemble `A` est égal à l'ensemble `B`, `False` sinon.

c. `disjoint(A,B)` : renvoie `True` si les ensembles `A` et `B` sont disjoints, `False` sinon.

6.2. Écrire en `python` les opérations ensemblistes suivantes :

a. `ajout(x,E)` : retourne le résultat de l'ajout de l'élément `x` à l'ensemble `E` ;

b. `union(A,B)` : retourne l'union de `A` et `B` ;

c. `intersection(A,B)` : retourne l'intersection de `A` et `B` ;

d. `retire(x,E)` : retourne le résultat de la suppression de l'élément `x` dans l'ensemble `E` ;

e. `diff(A,B)` : retourne l'ensemble $A \setminus B$

f. `diff_sym(A,B)` : retourne l'ensemble $A \Delta B$

4 Constructions de base

Python est muni d'un mécanisme de typage dynamique. Le type d'une variable est donc défini selon la valeur qui lui est affectée et peut changer en cas de nouvelle affectation. Il n'est donc pas nécessaire de déclarer un type avec la variable. On peut interroger le type d'un variable en utilisant la fonction `type(var)` ou préférentiellement (si la variable est susceptible d'être un "sous-type" du type testé) en questionnant avec la fonction `isinstance(var,type)`. Python dispose par ailleurs de la valeur particulière `None` qui peut être vue comme une constante utile pour exprimer l'absence de valeur (par exemple pour une fonction qui ne renvoie pas toujours de résultat).

Les types de base sont :

- `int` : entiers relatifs sur lesquels portent les opérateurs de calculs habituels (dont la puissance `**`) et aussi en particulier les opérateurs de quotient `//` et de reste `%` liés à la division euclidienne.
- `float` : nombre en virgule flottante
- `bool` : booléens `True` et `False`
- `str` : chaînes de caractères encadrées par des apostrophes ou guillemets, et sur lesquelles s'applique l'opérateur `+` de concaténation

Les objets créés avec ces types sont immuables. Cela signifie qu'après avoir créé un objet de ce type et lui assigné une valeur, on ne peut plus modifier cette valeur. Cela n'interdit pas pour autant d'affecter un nouvel objet de ce type à une variable existante.

```
>>>mot='toto'
>>>mot[1]
'o'
>>>mot[1]='a'
TypeError: 'str' object does not support item assignment
>>>mot='tata'
>>>mot
'tata'
```

Différentes structures de données

- `list` : une liste est une structure indexée et mutable. Doublons autorisés.

- `tuple` : un n-uplet est une structure indexée et immuable. Doublons autorisés.
- `set` : un ensemble est une structure non indexée et mutable. Pas de doublon autorisé.
- `dict` : un dictionnaire est une structure où l'indexation peut se faire par une chaîne de caractères. Chaque élément d'un dictionnaire s'apparente à un couple (clé,valeur) où la clé est la chaîne de caractères désignant l'indexation de la valeur dans le dictionnaire. Le dictionnaire est mutable. Pas de doublon autorisé.
- `array` : les tableaux possèdent les mêmes propriétés de base que les listes mais ils nécessitent l'import d'une bibliothèque pour être utilisés (`array` ou `numpy`)

Nous allons ci-après détailler les manipulations de listes. Un certain nombre de ces manipulations seront aussi valables sur les autres structures. Leurs propriétés respectives notamment en termes d'indexation, de mutabilité rendront néanmoins certaines opérations possibles ou non et des précautions devront être prises dans la copie ou l'appel à une variable à différents emplacements d'un programme selon la propriété de mutabilité.

4.1 Les listes

Quelques fonctionnalités pratiques

Définition d'une liste en extension Une liste peut-être définie en extension en faisant figurer entre crochets ses éléments séparés par des virgules, et la liste vide s'écrit [].

Longueur d'une liste La fonction `len` permet d'obtenir la longueur d'une liste.

```
>>>len([0,1,2])
3
```

Accès aux éléments d'une liste Lorsque `l` est une liste, l'expression `l[i]` désigne le `i`-ème élément de cette liste (avec $i > 0$). La position du premier élément d'une liste étant l'entier 0, une liste contenant n éléments permettra l'accès à un élément dont la position sera définie entre 0 et $n-1$. On peut également accéder aux éléments de la liste en indexant avec des nombres négatifs à partir de la droite (`l[-1]` désigne le dernier élément, `l[-2]` l'avant dernier...)

Dans le cas par exemple où une fonction renvoie une liste x, il peut parfois être pratique de déconstruire cette liste avec une instruction d'affectation du type

$$x_0, x_1, \ldots, x_{n-1} = x$$

qui permet d'affecter la valeur de $x[i]$ à chaque variable x_i.

Listes et affectations Lorsque `l1` est un nom de variable désignant une liste, la valeur associée au nom `l1` est la référence à la zone mémoire où sont stockés les éléments de la liste. Aussi, à l'issue de l'affectation `l2=l1`, du fait de la propriété de mutabilité des listes, la nouvelle valeur de `l2` est la référence à la zone mémoire où sont stockés les éléments de la liste `l1`, et si on modifie `l1` (resp. `l2`), alors on modifie également `l2` (resp. `l1`).

```
>>>l1=[4,7,1]
>>>l2=l1
>>>l1[1]=0
>>>l2
[4,0,1]
```

La liste `l1[:]` désigne une copie de tous les éléments de la liste `l1` et est stockée dans une zone mémoire qui lui est propre (on peut aussi créer une copie en utilisant la méthode `l1.copy()`). Une liste `l3` ainsi créée comme copie de `l1` ne sera donc pas modifiée en cas de modifications apportées sur `l1` (et vice-versa).

Listes en paramètres de fonction Puisque la valeur d'un nom désignant une liste correspond à une référence à la zone mémoire où sont stockés les éléments de cette liste, lorsqu'une fonction est appelée avec ce nom pour argument, c'est cette référence qui est transmise et utilisée dans le corps de la fonction. Aussi si le corps de cette fonction contient une instruction qui modifie la valeur d'un élément de cette liste, c'est la liste utilisée pour appeler la fonction qui est modifiée. L'exemple suivant illustre ce mécanisme :

```
def double_list(l):
      for i in range(0,len(l)):
            l[i]=2*l[i]
      return l
>>>l=[1,2,3,4]
>>>double_list(l)
[2,4,6,8]
>>>l
[2,4,6,8]
```

Pour ne pas modifier la liste utilisée lors de l'appel à cette fonction, il suffit d'appeler cette fonction avec une copie de la liste :

```
>>>l=[1,2,3,4]
>>>double_list(l[:])
[2,4,6,8]
>>>l
[1,2,3,4]
```

Sous-liste d'une liste Lorsque `l` est une liste, l'expression `l[i:j]` désigne la liste des éléments de `l` situés entre les positions de `i` et `j-1` dans `l` (si $j > i$, sinon c'est la liste vide). L'expression `l[:j]` est équivalente à l'expression `l[0:j]`, l'expression `l[i:]` est équivalente à l'expression `l[i:len(l)]` et l'expression `l[-k:]` avec $k > 0$ renvoie la liste des `k` derniers éléments de la liste `l`

Intervalle d'entiers Python permet de manipuler des intervalles d'entiers représentés par des listes d'entiers : `range(n,m)` désigne la liste $[n, n+1, \ldots, m-1]$ (lorsque $m > n$). On peut aussi utiliser la construction `range` avec un unique paramètre entier : `range(n)` désigne l'intervalle $[0, \ldots, n-1]$.

Itération sur les éléments d'une liste La construction `for` permet de répéter l'exécution d'un bloc d'instructions, délimité par l'indentation, en parcourant les éléments d'une liste. Par exemple, la fonction suivante permet de calculer la somme des éléments d'une liste de nombres.

```
def somme(l) :
      r=0
      for e in l:
            r=r+e
      return r
```

Définition en compréhension Python permet de définir une liste en compréhension. La construction `[e for x in s]`, où `x` est une variable de compréhension ou un n-uplet de variables, `s` est une séquence (une liste par exemple) et `e` est une expression (qui porte sur `x`), permet de construire la liste des valeurs successives de `e` obtenues à partir des valeurs successives de `x` issues du parcours de la séquence `s`.

```
>>>[x**2 for x in range(2,5)]
[4,9,16]
>>>[x+y for x,y in [(1,2),(3,4),(5,6)]
[3,7,11]
```

4.2 Les ensembles

Initialiser un ensemble On peut initialiser un ensemble vide en utilisant `set()`. Pour initialiser un ensemble avec des valeurs, on place ces valeurs entre deux accolades (l'ordre n'a pas d'importance puisque l'ensemble n'est pas ordonné). Attention la commande {} déclare un dictionnaire vide et non un ensemble.

Ajouter ou retirer des valeurs On peut utiliser la méthode `add` pour ajouter une valeur (`append` définie pour une liste n'est pas définie pour un ensemble) et la méthode `remove` pour en retirer une.

```
>>>radios_lucien=set()
>>>radios_lucien.add('nrj')
>>>radios_lucien
{'nrj'}
>>>radios_walid={'france info','france inter','fip' }
>>>radios_walid.remove('france info')
>>>radios_walid
{'fip','france inter'}
```

Remarque : L'ensemble renvoyé est présenté avec un ordre alphabétique par défaut

On ne peut ajouter que des valeurs immuables à un ensemble. Un `TypeError` apparaîtra si vous tentez d'ajouter une liste à un ensemble. A noter que si vous ajoutez un tuple, la valeur ajoutée aura le statut de tuple. Pour ajouter un ensemble de valeurs, on peut faire la réunion de l'ensemble initial avec l'ensemble contenant les valeurs à ajouter.

Des méthodes pour des opérations ensemblistes

On dispose pour les opérations ensemblistes des méthodes suivantes : `union`, `intersection`, `difference`, `symmetric_difference`

Quelques tests

- `A.isdisjoint(B)` Méthode pour interroger si les ensembles `A` et `B` sont disjoints
- 'france info' in radios_walid pour tester l'appartenance
- A.issubset(B) pour tester l'inclusion de A dans B.

4.3 La portée des variables

En Python, nous pouvons déclarer des variables n'importe où dans notre script : au début du script, à l'intérieur des boucles, au sein de nos fonctions, etc. L'endroit où on définit une variable va néanmmoins déterminer l'endroit où la variable va être accessible. L'expression "portée des variables" sert à désigner les différents espaces dans le script dans lesquels une variable est accessible. En Python, une variable peut

avoir une portée locale ou globale et on peut lister les variables enregistrées dans ces espaces de noms en utilisant les fonctions `globals()` et `locals()`.

Les variables définies dans une fonction sont appelées variables locales. Elles ne peuvent être utilisées que localement c'est-à-dire qu'à l'intérieur de la fonction qui les a définies. Appeler une variable locale depuis l'extérieur de la fonction qui l'a définie provoquera une erreur.

```
def ajoute(x):
      n=2
      l=[2]
      n=n+x
      l.append(x)
      print('n=',n)
      print('l=',l)
>>>ajoute(3)
n= 5
l= [2, 3]
>>>print(n)
NameError: name 'n' is not defined
>>>print(l)
NameError: name 'l' is not defined
```

Une variable définie au début du script sera enregistrée comme globale. Les fonctions exploiteront les variables définies localement, mais en dehors des fonctions ce seront les noms associés aux variables définies globalement qui seront exploités

```
n=1
l=[1]
def ajoute(x):
      n=2
      l=[2]
      n=n+x
      l.append(x)
      print('n=',n)
      print('l=',l)
>>>ajoute(3)
n= 5
l= [2, 3]
>>>print(n)
1
>>>print(l)
[1]
```

Si une variable est appelée dans une fonction sans qu'une variable n'y a été déclarée, le programme exploitera une variable globale portant ce nom uniquement si celle-ci est mutable et auquel cas l'exécution de la fonction pourra modifier la valeur de cette variable.

Si une variable globale portant ce même nom existe mais est immuable, l'exécution de la fonction renverra une erreur.

```
l=[1]
def ajoute(x):
      l.append(x)
      print('l=',l)
>>>ajoute(3)
l= [1, 3]
>>>print(l)
[1, 3]
```

```
n=1
def ajoute(x):
      n=n+x
      print('n=',n)
>>>ajoute(3)
local variable 'n' referenced before assignment
```

Dans le cas d'une variable immuable pour forcer la fonction à l'utiliser quitte à y assigner une nouvelle valeur on devra déclarer dans la fonction l'appel à cette

variable avec le mot clé `global`

```
n=1
def ajoute(x):
      global n
      n=n+x
      print('n=',n)
>>>ajoute(3)
n= 4
>>>print(n)
4
```