TP no 2

1 Ma première classe

Vous allez écrire (et faire des expériences avec) votre première classe.

Exercice 1. Écrivez un script contenant la définition d'une classe `SayHello` ayant les attributs méthodes suivants :

SayHello
language : : string welcomeStrings : : dict
sayHello() setLanguage(language : : string) languages() : : list of strings addHello(language : : string, message : : string)

— `language` est une chaîne de caractères (comme par exemple `fr` ou `en`) qui stocke la langue avec lequel il faudra dire bonjour.
— `welcomeStrings` est un dictionnaire qui stockera les chaînes de caractères avec lequel il dire bonjour. Par exemple, la valeur de ce dictionnaire pourra être
`{ 'fr': 'Bonjour', 'en': 'Hello' }`
— `sayHello()` affichera à l'écran le message de bonjour, dans la langue utilisée,
— `setLanguage(str)` permet de modifier la langue utilisée
— `languages()` permet de chercher quels sont le langages actuellement disponibles.
— `addHello(language, message)` permet d'ajouter ou de modifier un message de bienvenu dans un langue donnée.

Exercice 2. Testez votre script. Depuis un environnement interactif **python**

1. Créez un objet `sH` appartenant à la classe `SayHello`.
2. Utilisez l'objet `sH` pour afficher à l'écran `Bonjour`.
3. Utilisez l'objet `sH` pour afficher à l'écran `Bom dia` (mettez en ouvre toutes les étapes intermédiaires pour obtenir ce résultat)
4. Créez un autre `sH2` de la même classe. Utilisez l'objet `sH` pour afficher à l'écran `Hello` et l'objet `sH2` pour afficher à l'écran `Stay away from me`.
5. Ajoutez à `sH` des messages de bonjour dans toutes les langues suivantes : `fr`, `en`, `it`, `es`, `pt`. Vérifiez que les messages de bonjour dans ces langues ont été ajouté.
6. Créez deux instances différentes de la classe `SayHello`.
   — Comparez ces deux instances avec l'égalité usuelle.
   — Par rapport à quoi ce deux instances sont différentes.
   — Comment peut on vérifier, en utilisant l'interprète **python**, que ces deux instances sont différentes ?

2 Mon première héritage

Rappelez vous qu'une pile (`Stack`) est "essentiellement la même chose" qu'un file (`Fifo`) car ce deux type d'objets font essentiellement la même chose, sauf que au moment de retirer un élément, une pile enlève le dernier élément ajouté, une file enlève le dernier élément ajouté.
Dires que ces duex type d'objets sont essentiellement la même chose revient à dire que si on définit une classe `Stack` et une classe `Fifo`, ces classes auront le même méthodes :

SetAbstract
isEmpty() : : Bool pop() add(x : : any)

On dit donc que `Stack` et `Fifo` ont la même interface.

Exercice 3. Écrivez deux scripts contenant l'un la définition de la classe `Stack` et l'autre la définition de la classe et `Fifo`. L'interface de ces deux classes est donnée par la classe (abstraite) `SetAbstract` décrite ci-dessous.
Testez le bon fonctionnement des classes que vous avez défini : pour cela vous allez écrire un script `testStackFifo.py` separé avec au moins les tests suivants :
- Si on fait un `pop` sur une pile (`Stack`) vide, alors on obtient un message d'erreur envoyé par vous.
- De même comme auparavant, cette-fois ci pour une file (`Fifo`).
- Si on ajoute a et puis b à une pile (`Stack`) vide, et puis on fait un `pop`, alors on obtient b.
- Si on ajoute a et puis b à une file (`Fifo`) vide, et puis on fait un `pop`, alors on obtient a.

Vous avez certainement noté qu'il y a beaucoup de redondance entre cas deux classes. Nous allons *factoriser le code* en utilisant l'héritage.

Exercice 4. Définissez une classe `Set` dont l'interface est spécifiée ci-dessous :

Set
_state : : list
__init__() __str__() : : string isEmpty() : : Bool pop() : : any add(x : : any)

Définissez les classes `Stack` et `Fifo` comme héritant depuis la classe `Set`.
Questions :

1. Quels sont les méthodes à redéfinir ?
2. Quelle est la signification du tiret `_` dans `_state` ?
3. Peut on remplacer le nom de l'attribut `_state` par `state` ? Testez.
4. Quel est le désavantage de remplacer le nom de l'attribut `_state` par `state` ?
5. Peut on remplacer le nom de l'attribut `_state` par `__state` ? Testez.

Testez le bon fonctionnement des classes en réutilisant le script `testStackFifo.py` de l'exercice précèdent.

Masquer les détails l'implémetantation

Nombres complexes

Rappelez vous qu'un nombre complexe peut se représenter sous deux formes :

$$a + ib \qquad \text{où } a, b \in \mathbb{R}\,, \qquad\qquad re^{i\theta} \qquad \text{où } r \in \mathbb{R}_{\geq 0} \text{ et } \theta \in [-\pi, \pi].$$

a est la partie réelle, b la partie imaginaire, r son module, θ l'argument.

Exercice 5. Redigez un script **python** contenant la définition de deux classes `Complexe1`, `Complexe2` pour représenter un nombre complexe.
Ces classes auront les mêmes méthodes :

— `c.re()`, qui retourne la partie réelle d'un nombre complexe,
— `c.im()`, qui retourne la partie imaginaire d'un nombre complexe,
— `c.arg()` qui retourne l'argument d'un nombre complexe,
— `c.mod()` qui retourne le module d'un nombre complexe.

Dans `Complexe1`, un complexe aura, comme attributs d'instance, `self.re` et `self.im`.
Dans `Complexe2`, un complexe aura, comme attributs, `self.arg` et `self.mod`.

Exercice 6. Préfixez les attributs d'instance des ces deux classes par le double underscore (tiret du bas) , de façon a rendre ces deux attributs "privés". Pare exemple, `self.im` deviendra `self.__im`. Modifiez les deux classes de façon conséquente.
Via l'interpréteur, créez un nombre complexe, essayez de modifier ces attributs à la main.

Exercice 7. Ajoutez a toutes classes deux methodes constructeur `self.fromReIm(re,im)` et `self.fromModArg(mod,a`
Adaptez les constructeurs (méthodes `__init__`) des deux classes et ajoutez les méthodes `__str__`.

Exercice 8. Ajoutez au script deux fonction permettant l'une de faire l'addition de complexes, et l'autre de faire la multiplication.
Ces deux fonctions pourront avoir en paramètre des complexe de chaque classe.
Exemple :

```
>>> from complexe import *
>>> c1 = Complexe1(1,2)
>>> print(c1)
1.0 + i2.0
>>> c2 = Complexe2(1,3)
>>> print(c2)
1.0e^{i3.0}
>>> c3 = add(c1,c2)
>>> print(c3)
0.010007503399554585 + i2.1411200080598674
```

Mots comme chemin discrets, mots de Dyck

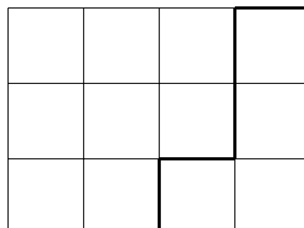
Notation : $\{ X, Y \}^*$ denote l'ensemble des mots sur l'alphabet $\{ X, Y \}$. Pour $\ell \in \{ X, Y \}$ et $w \in \{ X, Y \}^*$, soit $|w|_\ell$ le nombre d'occurrences de ℓ dans le mot w. Posons

$$P(n, m) = \{ w \in \{ X, Y \}^* \mid n = |w|_X, m = |w|_Y \} .$$

Donc $w \in P(n, m)$ ssi w a n occurrences de la lettre X et m occurrences de la lettre Y.
On pense à $w \in P(n, m)$ comme à un chemin discret de $(0, 0)$ vers (n, m), car on peut lire le mot de la gauche vers la droite, en faisant de mouvement dans le plan discret de cette facon

— on fait un mouvement de (a, b) vers $(a + 1, b)$ quand on lit la lettre X,
— et fait un mouvement de (a, b) vers $(a, b + 1)$ quand on lit la lettre Y.

Par exemple, le mot $XXYXYYX \in P(n, m)$ correspond au chemin discret



L'ensemble $P(n, m)$ est en bijection avec l'ensemble des fonctions croissantes (qui préservent l'ordre) de l'ensemble $\{1, \ldots, n\}$ vers l'ensemble $\{0, \ldots, m\}$. La bijection ψ est la suivante : pour un tel w et k avec $1 \leq k \leq n$, soit $w_{(X,k)}$ le plus petit préfixe de w contenant k occurrences de X. On pose alors

$$\psi(w)(k) = |w_{(X,k)}|_Y .$$

C'est-à-dire, $\psi(w)(k)$ est le nombre de lettres Y précèdent le k-ième occurrence de la lettre X. Par exemple, la table de $\psi(XXYXYYX)$ est la suivante

1	2	3	4
0	0	1	3

En exploitant cette bijection, on voit tout de suite que l'ensemble $P(n, m)$ possède beaucoup de structure algébrique, en particulier, il est un ensemble ordonné (avec l'ordre point à point des fonctions) et un treillis (le sup étant le max point à point, et le inf étant le min point à point).

Exercice 9. On souhaite ainsi manipuler sur ordinateur de tels chemins, en exploitant la bijection. Donc, la structure interne d'un tel chemin sera donné par une fonction—que l'on pourra représenter par une liste, ou par un dictionnaire, ou encore par un tableau.
Écrivez un script **python** contenant la définition d'une classe `Path` ayant l'interface suivante :

Path
n,m : int function
__init__(self,list) __str__(self) sup(self, other : Path) : Path inf(self, other : Path) : Path getValue(self, i : int) draw(self)

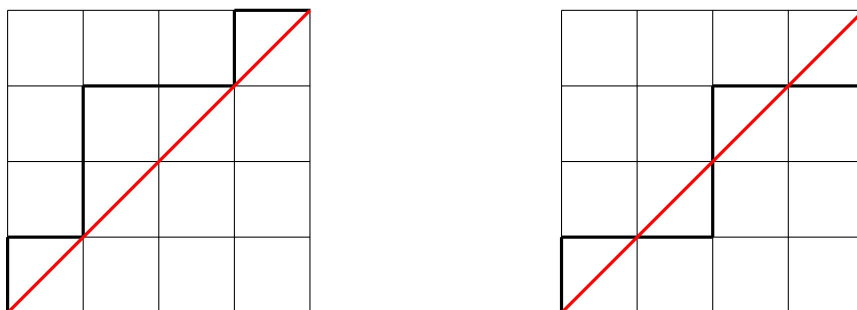
On commencera par prototyper cette définition : c'est-à-dire, on retardera l'écriture du code des méthodes à la fin fin du TP. Pour ce faire, on pourra exploiter l'instruction `pass` de **python**.
La méthode `getValue(i)` restituera la valeur de la fonction au point $i \in \{1, \ldots, m\}$.
La méthode `draw()` utilisera une biliotheque de **Python** pour afficher le chemin sous forme d'histogramme.

Exercice 10. Implémentez ensuite touts les méthodes dont l'algorithmique est triviale (`sup,inf,getValue,draw`).

Exercice 11. Attention : cette partie de l'exercice est celle plus subtile du point de vue algorithmique, mais ce n'est pas l'objectif du TP. Si vous ni arrivez pas dans une dizaine de minutes,passez à l'exercice suivant, plus important pour aujourd'hui.
Implémentez maintenant les méthodes spéciales `__init__` et `__str__`. La méthode constructeur sera paramétré par une chaîne de caractères, et se chargera de transformer cette chaîne de caractères dans sa représentation interne via une fonction croissante.
La méthode `__str__` (qui sera utilisée par la fonction `print`) transformera la représentation interne du chemin sous forme de fonction en chaîne de caractères.

Mots de Dyck

Un mot/chemin de Dyck de demi-longueur n est un mot $w \in P(n, n)$ qui reste toujours au dessus de la diagonale. Exemple (à la gauche) et contre-exemple (à la droite) :



Exercice 12. (Un peu d'héritage) Vous allez créer une classe (qui sera nommée `DyckPath`) pour le mot de Dyck en héritant depuis la classe `Path`.

Exercice 13. Redéfinissez la méthode constructeur pour vérifier que le chemin se trouve au dessus de la diagonale.