Dépôt BLIH: PYjour02
Répertoire de rendu : c'est précisée pour chaque exos.

pythonMiddle

Dans un sous répertoire **pythonMiddle**

Le python nous permets aussi de jouer avec les structures binaires avec le module *struct* et notamment les fonctions *pack* et *unpack*.

C'est sympa le binaire, on peut lire des paquets réseaux en python, générer des images etc...

Donc essentiel pour une piscine parsing.

Pour faire simple :

- **pack** permet de prendre des variables pythons et de les concaténer sous forme de buffer binaire.
- **unpack** permet de découper un buffer binaire sous forme de tuple python.

Les buffers binaires s'écrivent littéralement ainsi :

```
buf = b'\xCA\xFE\xBA\xBE'
```

La syntaxe **x** permet d'écrire la valeur des octets en hexadécimales dans ce qui ressemble à une string préfixé par **b**.

Toutefois, il existe 2 types pour les manipuler :

bytes pour les immutables (constante non modifiable).

bytearray pour les mutables (variable classique).

Faire un module « packman.py », tel que :

- une fonction **ushort_uint** qui prends un buffer et extrait un entier court non signé en big-endian, suivi d'un entier 32 bit non signé en big-endian.

```
$> python3.3
>>> import packman
>>> packman.ushort_uint(b'\x01\x42\x00\x01\x02\x03')
(322, 66051)
```

- une fonction **buf2latin** qui prends un buffer et en extrait la taille de la chaîne de caractères iso latin 1 et la chaîne qui suis.

```
$> python3.3
>>> import packman
>>> packman.buf2latin(b'\x00\x04G\xE9g\xe9')
(4, 'Gégé')
```

- une fonction **ascii2buf** qui prends un nombre variable de chaîne de caractères en paramètre et les transformes en un buffer tel que :
  - en premier nous avons le nombre d'éléments total dans le buffer en entier non signé 32 bit.
  - Ensuite chaque chaîne est concaténer et préfixer par sa taille (entier non signé 16 bit)

```
$> python3.3
>>> import packman
>>> packman.ascii2buf("I", "like", "the", "game")
bytearray(b'\x00\x00\x00\x04\x00\x01I\x00\x04like\x00\x03the\x00\x04game')
```

evalExpr

Dans un sous répertoire **evalExpr**

Encore un evalExpr !!!! Mais la on le fait correctement. Pas de polonaise inverse !!!

Attention les calculs sont de type entier. Allez voir la différence entre / et // en python.

Faire un script en python nommé 'evalExpr' capable de calculer des expressions arithmétiques simples et d'afficher le résultat.
Exemples : soit un fichier calcul.txt contenant

```
5*4
```

le programme s'utilise de la façon suivante :

```
# python3.3 evalExpr calcul.txt
20
```

A terme, vous devez pouvoir gérez ce genre d'expression : (chaque ligne est indépendante)

```
10
5*3
10-4-(64+3)*5
--7+8
-+---+++900*-(544)
```

variables :
Une fois les expressions classiques opérationnelles, on rajoute les variables.
On peut affecter une variable et l'utiliser :

```
iopi$ cat calc.txt
a=10
iopi$ python3.3 evalExpr calc.txt
10
iopi$ cat calc2.txt
c=4
b=c - 2
c-3
iopi$ python3.3 evalExpr calc2.txt
4
2
1
```

Conseils:

- Commencez par gérer * en écrivant une règle (ex : calculez 10, 10*10, 10*10*10,etc...). Comparez plusieurs échantillons d'entrée valide. Faites apparaître ce qui reste et ce qui apparaît entre 2 échantillons (ex : entre 10 et 10*10, etc...). Ajoutez / et %.
- Ajoutez + et - , cette nouvelle règle devrait utiliser la précédente.
- Ajoutez les (), ceci devrait vous faire modifier une des règles précédentes en faisant apparaître une troisième règle.
- Ajoutez le fait que «-- nombre» donne «nombre». Ajoutez les autres « + » unaire.
- Ajoutez les variables. Ceci devrait vous faire modifier les 2 règles les plus extrêmes du système.

L'affichage se déroule sur les expressions complètes pas sur les lignes :

```
iopi$ cat calc3.txt
4+4 5*5
2
+
1
iopi$ python3.3 evalExpr calc3.txt
8
```

```
25
3
```