

Examen Tutorat

Note : pour cet examen il est **impératif** que vous ayez configuré votre éditeur de texte pour utiliser 4 espaces par niveau d'indentation car du code utilisant cette norme vous est fournis. Si vous ne l'avez pas fait référez vous au TP0 d'introduction à Linux pour savoir comment faire. Si vous ne configurez pas votre éditeur correctement, le code que vous écrirez **ne marchera pas**.

Décompressez l'archive sur votre bureau, et renommez le dossier PRENOM_NOM_TUTOPROG suivant votre nom et prénom.

Note : n'utilisez pas d'espace ou de caractères accentués dans le nom du dossier. À la place des espaces utilisez des underscores (tiret du bas en Français)

Dans ce dossier, vous trouverez trois fichiers : **exam.py**, **runtests.py** et **colors.py**.

Pour cet examen, vous devez **directement modifier** le fichier **exam.py**, **sans rien enlever** de ce qui est déjà écrit dedans. **Vous ne devez pas modifier les autres fichiers**.

Vous trouverez dans **exam.py** des headers de fonctions accompagnés de chaînes de documentation. Les chaînes de documentations comportent des exemples de ce qui devrait se passer si vous invoquez les fonctions directement depuis le Shell de Python.

Ces exemples vont aussi servir à tester automatiquement vos implémentations de chaque fonction.

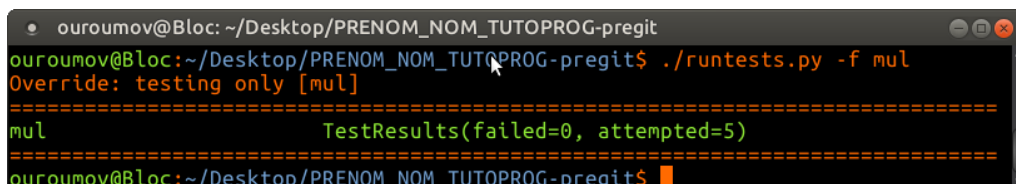
Dans chaque cas vous devez compléter le code de la fonction (body) et ensuite exécuter le fichier fournis **runtests.py** pour tester votre code.

Tout ce qui est écrit dans exam.py est **en Anglais**, mais la suite de ce document décrit **en Français** ce que chacune des fonctions doit faire.

Votre mission est de compléter le fichier **exam.py** jusqu'à ce que tous les tests réussissent. Faites une fonction, testez, puis si les tests passent passez à la fonction suivante.

Pour exécuter les tests, ouvrez un terminal et déplacez vous dans le dossier où se trouvent les fichiers, puis exécutez le fichier **runtests.py**.

Note : Si vous êtes bloqués par une fonction, vous pouvez tester juste une fonction en utilisant l'argument **-f fonction** sur la ligne de commande de **runtests.py** :



```
ouroumov@Bloc: ~/Desktop/PRENOM_NOM_TUTOPROG-pregit
ouroumov@Bloc:~/Desktop/PRENOM_NOM_TUTOPROG-pregit$ ./runtests.py -f mul
Override: testing only [mul]
=====
mul                               TestResults(failed=0, attempted=5)
=====
ouroumov@Bloc:~/Desktop/PRENOM_NOM_TUTOPROG-pregit$
```

Certaines fonctions sont regroupées en suite logique :

[6, 7, 8]

[9, 10, 11, 12]

[13, 14, 15, 16, 17]

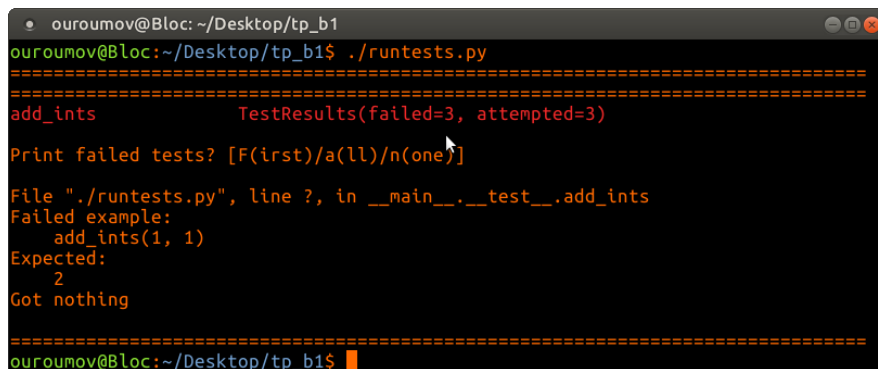
Les fonctions [2, 3, 4, 5] sont complètement indépendantes.

1) add_ints (*)

```
def add_ints(a, b):  
    ''' (int, int) -> int  
  
    Return the sum of a and b.  
  
    >>> add_ints(1, 1)  
    2  
    >>> add_ints(1, 3)  
    4  
    >>> add_ints(1, -6)  
    -5  
    '''  
    return a + b
```

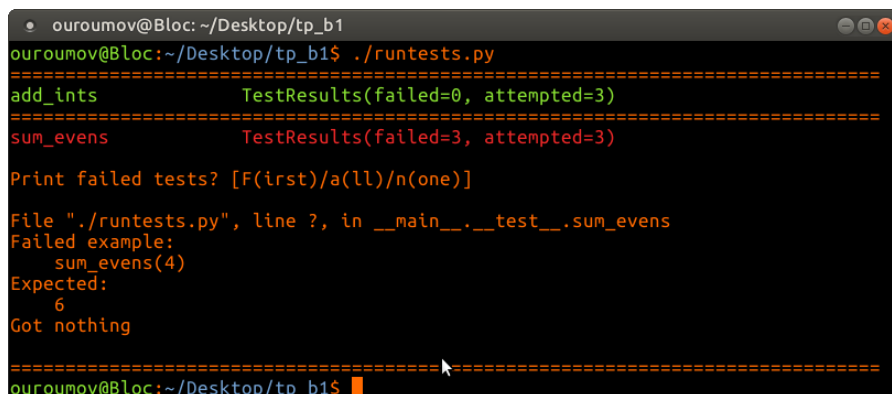
Cette fonction vous est fournie à titre d'exemple. Elle prends deux entiers en arguments et renvoie leur somme. Cette fonction n'est pas là pour servir d'exercice de programmation, juste pour vous permettre de voir par vous même comment fonctionne ce TP.

Si vous enlevez la ligne où se trouve le return puis exécutez **runtests.py** les tests pour cette fonction vont échouer, et le programme **runtests.py** va vous proposer des options. Si vous enfoncez simplement la touche entrée l'option par défaut affiche le premier test qui a échoué et vous verrez cela :



```
ouroumov@Bloc: ~/Desktop/tp_b1  
ouroumov@Bloc:~/Desktop/tp_b1$ ./runtests.py  
=====  
add_ints                TestResults(failed=3, attempted=3)  
=====  
Print failed tests? [F(irst)/a(ll)/n(one)]  
File "./runtests.py", line ?, in __main__.__test__.add_ints  
Failed example:  
    add_ints(1, 1)  
Expected:  
    2  
Got nothing  
=====  
ouroumov@Bloc:~/Desktop/tp_b1$
```

Après avoir remis la ligne return, et en supposant que vous l'avez fait correctement, vous devriez voir ça :



```
ouroumov@Bloc: ~/Desktop/tp_b1  
ouroumov@Bloc:~/Desktop/tp_b1$ ./runtests.py  
=====  
add_ints                TestResults(failed=0, attempted=3)  
=====  
sum_evens                TestResults(failed=3, attempted=3)  
=====  
Print failed tests? [F(irst)/a(ll)/n(one)]  
File "./runtests.py", line ?, in __main__.__test__.sum_evens  
Failed example:  
    sum_evens(4)  
Expected:  
    6  
Got nothing  
=====  
ouroumov@Bloc:~/Desktop/tp_b1$
```

Cela vous indique que add_ints a passé trois tests et que zéro test ont échoué, puis cela vous indique que les tests de la fonction suivante dans le fichier ont échoués, ce qui est normal.

2) f1 (*)

```
def f1(n):  
    ''' (int) -> int  
  
    Returns 4 times n square plus n plus 9.  
  
    >>> f1(1)  
    14  
    >>> f1(2)  
    27  
    >>> f1(100)  
    40109  
    '''
```

Cette fonction prends en argument un entier « n » et doit renvoyer : $4n^2 + n + 9$

Note : la première ligne de la chaîne de documentation [(int) -> int] est ce qui s'appelle un contrat de type, elle est là pour vous indiquer le type des paramètres et le type de la valeur de retours attendue. Ici la fonction attends qu'on lui passe un entier en argument et doit renvoyer un entier.

3) f2 (*)

```
def f2(x, y):  
    ''' (float, float) -> float  
  
    Returns x times y over the sum of x, y and 2.  
  
    >>> "%.04f" % f2(1, 8.5)  
    '0.7391'  
    >>> "%.04f" % f2(9, 2.0)  
    '1.3846'  
    >>> "%.04f" % f2(1.0, 0)  
    '0.0000'  
    '''
```

Cette fonction prends en argument deux float « x » et « y » et doit renvoyer : $\frac{x * y}{x + y + 2}$

4) mul (**)

```
def mul(l):  
    ''' (list of int) -> int  
  
    Return the product of every element in the list.  
  
    >>> mul([2, 2])  
    4  
    >>> mul([1, 2, 2])  
    4  
    >>> mul([1, 2, 3])  
    6  
    >>> mul([1, 9, 6, 4, 2, 6, 1, 8, 9, 2])  
    373248  
    >>> mul([0, 9, 6, 4, 2, 6, 1, 8, 9])  
    0  
    '''
```

Cette fonction prends en argument une liste d'entiers et doit renvoyer le produit de tous les entiers de la liste.

5) sum_evens (**)

```
def sum_evens(n) :  
    ''' (int) -> int  
  
    Return the sum of every even number from 0 to n (inclusive).  
  
    >>> sum_evens(4)  
    6  
    >>> sum_evens(5)  
    6  
    >>> sum_evens(6)  
    12  
    '''
```

Cette fonction prends un entier « n » en argument, et doit renvoyer la somme de tous les entiers pairs entre 0 et n (inclus).

6) surround (*)

```
def surround(c, s) :  
    ''' (str, str) -> str  
  
    Returns s surrounded on both sides by c.  
  
    >>> surround('+', 'test')  
    '+test+'  
    >>> surround('!', 'B')  
    '!B!'  
    '''
```

Cette fonction prends en argument un caractère « c » (chaîne de longueur 1) et une chaîne « s » et doit renvoyer « s » entourée de chaque côté par un caractère « c ».

Note : vous pouvez concaténer des chaînes de caractères avec l'opérateur + :

```
>>> s = 'what'  
>>> 'lol' + s  
'lolwhat'
```

7) center_in_field (****)

```
def center_in_field(c, s, w):
    ''' (str, str, int) -> str

    Returns s centered in a field of characters c with length w.
    If there's an odd number of characters in the field, add the extra character
    on the right.

    >>> center_in_field('.', 'lol', 5)
    '.lol.'
    >>> center_in_field('.', 'lol', 6)
    '.lol..'
    >>> center_in_field('+', 'lol', 7)
    '++lol++'
    >>> center_in_field(' ', 'lol', 8)
    ' lol  '
    '''
```

Cette fonction est la première fonction un peu corsée de ce TP.

Elle prends en paramètre un caractère « c », une chaîne « s », et un entier « w » et doit renvoyer « s » centrée dans un champ remplis de caractères « c » et d'une largeur de « w ». Si le nombre de caractères à rajouter est impair, alors le caractère de trop doit être ajouté à droite.

Note : considérez que pour cette fonction, « w » est toujours au moins égal à la longueur de la chaîne « s », vous n'avez pas besoin de vérifier cela dans votre code. Vous pouvez obtenir la longueur de la chaîne « s » avec la fonction Python **len** : [**len**('lol') → 3]

Note : rappelez vous que vous pouvez multiplier des chaînes de caractères de la manière suivante :

```
>>> "B" * 4
'BBBB'
```

8) comment_block (****)

```
def comment_block(c, s, w):
    ''' (str, str, int) -> str

    >>> comment_block('#', 'test', 8)
    '#####\n# test #\n#####'
    >>> print comment_block('#', 'test', 10)
    #####
    # test #
    #####
    '''
```

Cette fonction doit créer un block de commentaires correspondant à un caractère « c » qui est le caractère de commentaire (en Python c'est « # »), une chaîne « s » qui est la chaîne à commenter et une largeur de block « w ».

Cette fonction est difficile à écrire, vous devez obligatoirement utiliser les fonctions **center_in_field** et **surround** dans le code de cette fonction, et gardez à l'esprit que vous pouvez littéralement écrire un retour à la ligne en utilisant « '\n' » de la manière suivante :

```
>>> s = "test" + "\n" + "test"
>>> print s
test
test
```

9) is_nucleotide (*)

```
def is_nucleotide(c):  
    ''' (str) -> Boolean  
  
    Return True if c in one of 'ATCG'  
  
    >>> is_nucleotide('A')  
    True  
    >>> is_nucleotide('B')  
    False  
    >>> is_nucleotide('l')  
    False  
    >>> is_nucleotide('!')  
    False  
    >>> is_nucleotide('G')  
    True  
    >>> is_nucleotide('C')  
    True  
    >>> is_nucleotide('T')  
    True  
    '''
```

On va faire un peu de biologie avec cette fonction : elle prends un caractère et dit si ce caractère correspond à un des 4 nucléotides de base A, T, C ou G.

Note : « a » ne correspond pas à « A » et n'est pas un nucléotide.

Note : rappelez vous que vous pouvez tester si une chaîne est dans une autre de la manière suivante :

```
>>> 'o' in 'lol'  
True  
>>> 'z' in 'lol'  
False
```

10) is_valid_dna_sequence (*)

```
def is_valid_dna_sequence(s):  
    ''' (str) -> Boolean  
    Return True if s is a valid dna sequence (contains only nucleotides)  
  
    >>> is_valid_dna_sequence('AT')  
    True  
    >>> is_valid_dna_sequence('ATF')  
    False  
    >>> is_valid_dna_sequence('TTATCCCG')  
    True  
    >>> is_valid_dna_sequence('TTlTCCCG')  
    False  
    '''
```

Cette fonction doit vérifier si la chaîne « s » est une séquence de nucléotides, c'est à dire qu'elle contient uniquement des nucléotides. Cette fonction doit impérativement utiliser la fonction précédente **is_nucleotide**.

11) get_complement (*)

```
def get_complement(c):  
    ''' (str) -> str  
  
    Return the complement of nucleotide nuc.  
  
    >>> get_complement('A')  
    'T'  
    >>> get_complement('T')  
    'A'  
    >>> get_complement('C')  
    'G'  
    >>> get_complement('G')  
    'C'  
    '''
```

Cette fonction prends un nucléotide en argument et doit renvoyer son complément. Le complément de « A » est « T », celui de « C » est « G » et inversement.

Pour cette fonction on suppose que l'argument passé est un nucléotide, vous n'avez pas besoin de vérifier cela.

12) get_complement_sequence (*)

```
def get_complement_sequence(s):  
    ''' (str) -> str  
  
    Return the complementary sequence of DNA sequence s.  
  
    >>> get_complement_sequence('AT')  
    'TA'  
    >>> get_complement_sequence('TA')  
    'AT'  
    >>> get_complement_sequence('CCGCTAT')  
    'GGCGATA'  
    '''
```

Cette fonction prends une séquence de nucléotides et doit renvoyer le complément de cette séquence.

Pour cette fonction on suppose que l'argument passé est une séquence valide de nucléotide, vous n'avez pas besoin de vérifier cela.

13) get_number_from_letter (**)

```
def get_number_from_letter(c):  
    ''' (str) -> int  
    Return the number corresponding to letter c  
  
    >>> get_number_from_letter('A')  
    0  
    >>> get_number_from_letter('B')  
    1  
    >>> get_number_from_letter('Z')  
    25  
    >>> get_number_from_letter('X')  
    23  
    '''
```

Avec cette fonction on laisse tomber la biologie et on commence une mini section sur les bases de la crypto.

Cette fonction doit faire correspondre à une lettre en majuscule (de A à Z) un numéro.
A est la première lettre et doit avoir le numéro 0, et Z est la dernière et doit avoir le numéro 25.

Note : vous devez utiliser la fonction python **ord** (testez la dans le shell) pour convertir une lettre en entier, mais cette fonction a un décalage de 65 car le caractère « A » est le [65ème caractère du code ascii](#).

14) get_letter_from_number (**)

```
def get_letter_from_number(n):  
    ''' (int) -> str  
  
    >>> get_letter_from_number(0)  
    'A'  
    >>> get_letter_from_number(2)  
    'C'  
    >>> get_letter_from_number(24)  
    'Y'  
    >>> get_letter_from_number(25)  
    'Z'  
    '''
```

Cette fonction est l'inverse de la fonction précédente : elle doit prendre un nombre de 0 à 25 et renvoyer la lettre qui correspond.

Il faut au préalable appliquer le même décalage de 65 mais dans l'autre sens, puis utiliser ensuite la fonction Python **chr** (que vous pouvez aussi tester dans le shell).

15) shift_mod26 (***)

```
def shift_mod26(n, shift):  
    ''' (int) -> int  
  
    Return n shifted by "shift" in the [0-25] space  
  
    >>> shift_mod26(0, 2)  
    2  
    >>> shift_mod26(25, 2)  
    1  
    >>> shift_mod26(1, 4)  
    5  
    >>> shift_mod26(1, 4050)  
    21  
    '''
```

Cette fonction prends un entier « n » en argument et doit le décaler dans l'espace [0-25], si le décalage de l'entier fait dépasser la borne 25, alors le nombre doit revenir au début (Si on passe 25 avec un shift de 1, ça doit donc revenir à 0)

16) encrypt_char (*)

```
def encrypt_char(c, shift):  
    ''' (str) -> str  
  
    Return an encrypted version of char "c" : it is shifted by "shift" places.  
  
    >>> encrypt_char('A', 1)  
    'B'  
    >>> encrypt_char('A', 2)  
    'C'  
    >>> encrypt_char('Z', 2)  
    'B'  
    '''
```

Cette fonction prends un caractère en argument et « chiffre » ce caractère par un simple décalage. Le décalage est la « clef » utilisé pour le chiffrement :

« A » avec un décalage de 1 produit la version chiffrée de ce caractère : « B »

Note : pour cette fonction vous devez utiliser les fonctions **get_number_from_letter**, **shift_mod26** et **get_letter_from_number** écrites précédemment.

17) encrypt_string (**)

```
def encrypt_string(s, shift):  
    ''' (str) -> str  
  
    >>> encrypt_string("TEST", 1)  
    'UFTU'  
    >>> encrypt_string("THIS IS A TEST", 1)  
    'UIJT JT B UFTU'  
    '''
```

Finalement, cette fonction doit prendre une chaîne de caractère et retourner une version chiffrée.

Cette fonction doit utiliser la fonction précédente **encrypt_char**.

Tout caractère qui n'est pas une lettre majuscule (Entre A et Z) doit être ajouté tel quel à la chaîne renvoyée.

Bonne année tout le monde !