Notebook séance 17

Gestion des exceptions en Python.

Les erreurs

Lorsqu'on diviste par zéro, qu'on accède à un élément d'une liste qui n'existe pas, qu'on accède à la clé d'un dictionnaire qui n'existe pas des erreurs sont levées par Python

```
3/0
```

```
l = [1,2,3]
l[10]
```

```
d = {'a' : 1, 'b' : 3}
d['c']
```

Les erreurs qui arrivent portent des noms : `ZeroDivisionError`, `IndexError`, `KeyError`. Ce sont ce qu'on appelle des **exceptions**.

Une première manière de gérer soi-même des erreurs : utilisation de `assert`

Par exemple vérifier que lorsqu'on passe un numéro en paramètre d'une fonction, le numéro est dans l'intervalle de définition. On pourra faire cela grâce aux instructions `assert` qui permettent de vérifier qu'une expression booléenne est vraie. Si l'expression est vraie, il ne se passe rien, sinon un message d'erreur est affiché et le programme s'interrompt.

```
num = 2
assert num < 3, "{} n'est pas inférieur à 3".format(num)
# rien ne se passe
print("je suis passé à l'instruction suivante")
```

```
num = 5
assert num < 3, "{} n'est pas inférieur à 3".format(num)
# une erreur est déclenchée
print("je suis passé à l'instruction suivante")
```

```
def verifDNA(s):
    for c in s:
        if c not in ['a','t','c','g']:
            return False
    return True
```

```
assert verifDNA("atcgagagtagag") , "n'est pas composé de atcg"
```

```
assert verifDNA("atcgagnnqhhjjjsuagtagag") , "n'est pas composé de atcg"
```

Remarquez que là aussi cela *déclenche* une exception (on dit aussi *lever* une exception). L'exception `AssertError` est déclenchée.

Cela fonctionne bien pour les cas où il y a des assertions à vérifier, c'est-à-dire des propriétés qui doivent être vraies dans le programme, autrement dit ce n'est pas sensé arriver.

L'inconvénient c'est que cela interrompt le programme. Autrement dit dans le cas où on veut demander à un utilisateur de saisir quelquechose au clavier, on est bloqué.

```
# on souhaite que l'utilisateur réponde Y ou N
reponse = input("Entrez Y/N : ")
assert reponse in ['Y','N'], "On attend Y ou N"
print("La réponse est {}".format(reponse))
```

```
# on souhaite que l'utilisateur saisisse un entier positif
nb = int(input("Entrez un nombre entier : "))
assert nb >= 0, "Un entier positif est attendu"
print("Nombre saisi : {}".format(nb))
```

```
reponse_valide = False
while not reponse_valide:
    reponse = input("Entrez Y/N : ")
```

```
    reponse_valide = reponse in ['Y','N']
#    if reponse in ['Y','N']:
#        reponse_valide = True
#    else:
#        reponse_valide = False
print("La réponse est {}".format(reponse))
```

Mais ce schéma ne fonctionne pas toujours.

```
# on souhaite que l'utilisateur saisisse un entier positif
reponse_valide = False
while not reponse_valide:
    reponse = int(input("Entrez un nombre entier positif : "))
    reponse_valide = reponse >= 0
print("La réponse est {}".format(reponse))
```

Une seconde manière de gérer les erreurs : rattraper les exceptions et réaliser un traitement approprié

Les attraper

C'est d'essayer de faire quelque chose et de prévoir un plan B en cas d'échec. On utilise la structure de contrôle :

```
try:
    # plan A
    intsruction 1
    intsruction 2
    instruction 3 # ->> declenche une exception
    instruction 4 # non evaluee
    instruction 5 # non evaluee
except:
    # plan B
    instruction B1 # evaluee apres l'echec de l'insctruction 3
    instruction B2
    instruction B3
```

```
l = [ i for i in range(10) ]
```

```
# on souhaite que l'utilisateur saisisse un entier positif
reponse = None
try:
    reponse = int(input("Entrez un nombre entier : "))
    print("La réponse est {}".format(reponse))
except:
    print("Merci d'entrer un entier")
```

```
# on souhaite que l'utilisateur saisisse un entier positif
reponse = None
while reponse is None:
    try:
        reponse = int(input("Entrez un nombre entier : "))
        print("La réponse est {}".format(reponse))
    except:
        print("Merci d'entrer un entier")
```

```
# on souhaite que l'utilisateur saisisse un entier positif
reponse_valide = False
while not reponse_valide:
    try:
        reponse = int(input("Entrez un nombre entier positif : "))
        reponse_valide = reponse > 0
        if reponse_valide:
            print("La réponse est {}".format(reponse))
        else:
            print("Merci d'entrer un entier positif (i.e. > 0)")
```

```
    except:
        print("Merci d'entrer un entier positif")
```

```
# on souhaite que l'utilisateur saisisse un entier positif
reponse_valide = False
while not reponse_valide:
    try:
        reponse = int(input("Entrez un nombre entier positif : "))
        reponse_valide = reponse > 0
        assert reponse > 0, "Merci d'entrer un entier positif (i.e. > 0)"
        print("La réponse est {}".format(reponse))
    except:
        print("Merci d'entrer un entier positif")
```

Mais il faut parfois être plus précis, et avoir plusieurs plans B, suivant pourquoi l'erreur est levée.

```
reponse_valide = False
while not reponse_valide:
    try:
        e1 = int(input("Entrez un nombre entier : "))
        e2 = int(input("Entrez un nombre entier : "))
        reponse_valide = True
        print("{} / {} = {}".format(e1,e2,e1/e2))
        ### ....
        l = []
        ###
        l[1] = e1/e2 # declenche une erreur
    except Exception as e:
        print("Merci d'entrer un entier")
        print(repr(e))
```

```
int(input("Entrez un nombre entier : "))
```

```
4/0
```

```
l = []
l[1] = 4.0
```

```
reponse_valide = False
while not reponse_valide:
    try:
        e1 = int(input("Entrez un nombre entier : "))
        e2 = int(input("Entrez un nombre entier : "))
        reponse_valide = True
        print("{} / {} = {}".format(e1,e2,e1/e2))
    except ValueError:
        print("Merci d'entrer un entier")
    except ZeroDivisionError:
        print("Merci d'entrer un entier non nul pour le dénominateur")
```

Les déclencher

On peut lever une exception soi-même avec `raise`

```
NB_EXONS = 8
exons = [ 'e{}'.format(i+1) for i in range(NB_EXONS)]
exons
```

```
def get_exon(number):
    """
    Paramaters
    ----------
    number : int
        1 <= number <= NB_EXONS
```

```
    """
    return exons[number-1]

get_exon(1)

get_exon(10)

get_exon(0)

NB_EXONS = 8
exons = [ 'e{}'.format(i+1) for i in range(NB_EXONS)]

def get_exon(number):
    """
    Paramaters
    ----------
    number : int
        1 <= number <= NB_EXONS

    Exceptions:
    -----------
    ValueError when number < 1 or number > NB_EXONS
    """
    if number < 1 or number > NB_EXONS:
        raise ValueError("Le numéro d'exon doit être compris entre 1 et {}".format(NB_EXONS))
    return exons[number-1]

get_exon(0)

reponse_valide = False
while not reponse_valide:
    try:
        e1 = int(input("Entrez un numéro d'exon : "))
        print(get_exon(e1))
        reponse_valide = True
    except ValueError as e:
        print(e)
```

Les exceptions connues

```
import builtins
dir(builtins)

help(dir)
```

Exercices

Exercice 1

Reprendre la fonction de lecture d'un fichier FASTA et la modifier pour qu'elle lève l'exception `ValueError` lorsque la séquence contient d'autres lettres que 'ATCG', et l'exception `SyntaxError`si le fichier n'est pas au format FASTA (i.e. la première ligne n'est pas du type `> commentaire`. Puis écrire un programme principal qui affiche la taille de la séquence passée en argument mais aussi : - affiche un message à l'utilisateur lorsque le nom de fichier fournit en paramètre n'existe pas - affiche un message lorsque le fichier est malformé - affiche un message lorsque la séquence n'est pas une séquence de 'ATCG'

```
set("ATCGGGCAGA") == {'A', 'G', 'C', 'T'}

set("ATCGGGCAGA")
```

Exercice 2

Les expressions régulières permettent la modélisation de motifs que l'on peut rechercher dans une chaîne de caractères. Voir cette partie du cours pour la syntaxe des expressions.

Par exemple l'expression "A.." permet de rechercher toutes les occurrences des mots de 3 lettres débutant par A.

Pour trouver toutes les occurrences d'une expression régulière en Python on peut utiliser la fonction `findall`.

```
import re
s = "ATCGGGCAGA"
regex = re.compile("A..")
resultat = regex.findall(s)
resultat
```

Quelle est l'exception déclenchée lorsqu'on passe à la fonction `compile` une expression malformée (par exemple "A)" est une expression malformée) ?

Ecrire une fonction `nb_occurrences` :

```
def nb_occurrences (r,s):
    """
    Parameters:
    ----------
    r : String
        regular expression
    s : String
        the string in which the regular expression is searched for
    Returns:
    -------
    int
        the number of occurrences of r in s
    Exceptions:
    ----------
    TODO
    """
```

Ecrire un main qui affiche à l'utilisateur le nombre d'occurrence d'un motif passé en 1er argument dans une chaîne passée en second argument et affiche un message explicite en cas de problème.

Créer ses propres exceptions

Pour avoir une gestion des exceptions plus fine, il est donc possible de créer ses propres exceptions, au delà de celles qui existent déjà.

Pour faire cela c'est assez simple, on va créer une nouvelle classe d'exception qui sera basée sur le modèle de la classe `Exception`.

```
class AlphabetError (Exception):
    pass

def verif (letter):
    if letter not in ['A','T','C','G']:
        raise AlphabetError()

verif('A')

verif('H')
```

Mixer les deux : déclenchement et récupération

Il est aussi possible de récupérer une expeption et d'en redéclencher une. Cela a deux utilités : - déclencher une exception plus appropriée - faire un traitement avant de redéclencher l'exception

```
class InputError(Exception):
    pass

def diviser(e1,e2):
    try:
        return int(e1)/int(e2)
    except ValueError:
        raise InputError("Un entier est attendu")
```

```
    except ZeroDivisionError:
        raise InputError("Un entier non nul est attendu pour le dénominateur")

try:
    e1 = input("Entrez un nombre entier : ")
    e2 = input("Entrez un nombre entier : ")
    print(diviser(e1,e2))
except InputError as e:
    print("Erreur des nombres en entrées : {}".format(str(e)))
```

Exercices

Exercice 3

Dans l'exercice de modélisation d'un gène, ajouter une méthode `get_exon` qui retourne le n-ème exon (n étant passé en paramètre) de sorte qu'elle déclenche une exception `UnknownExonError` qu'on définira lorsqu'on demande un exon dont le numéro n'existe pas.

Exercice 4

Reprendre la classe `DNA`, repérer les endroits où il peut y avoir des erreurs et proposer des nouvelles exceptions puis les intégrer dans le code de la classe (et les commentaires).