Exceptions
Corrigé

Objectifs
— Comprendre les exceptions

Remarque : L'exercice 4 est optionnel.

1 Comprendre le mécanisme d'exception

Exercice 1 : Comprendre le mécanisme d'exception

Dans cet exercice, on considère le programme suivant.

```
'''Comprendre les exceptions.'''

def ecrire(objet):
    '''afficher l'objet sans retour à la ligne'''
    print(objet, end='')

def f2(p: str) -> None:
    ecrire('<')
    if p is None:
        raise ValueError('must not be None')
    if not isinstance(p, str):
        raise TypeError('str expected but type is ' + str(type(p)))
    ecrire(p[0])
    ecrire(p[1])
    ecrire('>')

def f1(p1: str) -> None:
    ecrire('[')
    try:
        ecrire('(')
        f2(p1)
        ecrire(')')
    except ValueError:
        ecrire('V')
    except TypeError:
        ecrire('T')
    finally:
        ecrire('F')
    ecrire(']')
```

1.1. Pour chacun des appels suivants, sur papier, indiquer ce qui sera affiché :

```
f1('un')
f1(None)
f1(10)
f1('x')
```

Solution :

```
>>> f1('un')
[(<un>)F]
>>> f1(None)
[(<VF]
>>> f1(10)
[(<TF]
>>> f1('x')
[(<xFTraceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/cregut/projects/ens/python/exercices/comprendre_exception.py", line 21, in f1
    f2(p1)
  File "/home/cregut/projects/ens/python/exercices/comprendre_exception.py", line 14, in f2
    ecrire(p[1])
IndexError: string index out of range
```

On note que F est toujours affiché. En effet, dès que l'exécution a franchi un `try`, le bloc `finally` correspondant est toujours exécuté, qu'une exception se produise ou non, qu'elle soit récupérée ou non.

Bien noter que l'instruction `raise` n'affiche rien. Elle se contente de lever une exception, généralement créée dans l'expression associée avec un message (une chaîne) qui explique pourquoi l'exception est créée. Ce message n'est pas affiché mais pourra être affiché par celui qui récupèrera l'exception (l'appelant ou l'un de ses appelants).

1.2. Comment faire pour récupérer toutes les exceptions ?

Solution : On peut faire `except Exception`. Ceci récupère presque toutes les exceptions. Si on veut vraiment toutes les récupérer, il faut remplacer `Exception` par `BaseException`.

Attention : Il n'est pas recommandé de récupérer toutes les exceptions. En effet, on ne saura pas précisément quelle exception s'est produite ni pourquoi ; on ne saura donc pas quoi faire pour corriger le problème signalé par cette exception.

Remarque : On peut faire `except:` (sans préciser de type d'exception). C'est comme si on avait mis `BaseException`.

2 Quelques fonctions

Exercice 2 : Chaîne vers entier naturel

À l'image de la « fonction » `int(str)`, on veut écrire une fonction `natural(str)` qui retourne l'entier naturel correspondant à la chaîne de caractères passée en paramètres. Par exemple, `natural("421")` retournera l'entier 421.

Si la chaîne est autre chose qu'un entier, une erreur sera signalée via l'exception `ValueError` (exception levée par `int(str)` dans les mêmes conditions) et `SignError` si la chaîne correspond à un entier strictement négatif. Par exemple `natural("abc")` laissera se propager l'exception `ValueError` et `natural("-13")` laissera se propopager `SignError`.

2.1. Définir l'exception `SignError` (fichier `natural.py`).

Solution :

```
class SignError(Exception):
    pass
```

2.2. Écrire la fonction `natural(str)` (fichier `natural.py`). On s'appuiera sur la fonction `int(str)`. On utilisera le fichier `test_natural.py` pour la tester.

Solution :

```
def natural(chaine):
    """
    Retourner l'entier naturel qui correspond à chaine.

    Paramètres
        chaine: une chaîne de caractères

    Retour
        L'entier naturel qui correspond à chaine

    Exception
        ValueError si chaine n'est pas en entier
        SignError si l'entier est négatif
    """
    assert isinstance(chaine, str)

    entier = int(chaine)
    if entier < 0:
        raise SignError("Non positiv integer: " + chaine)
    return entier
```

Une exception est le moyen de signaler à l'appelant que l'on a pu réaliser ce qui était demandé. Ici, si la chaîne ne correspond pas à un entier, on ne pourra pas fournir l'entier naturel correspondant. On le signale via l'exception `ValueError`. De la même manière, si la chaîne correspondant à un entier strictement négatif, ce n'est pas une entier naturel. On le signale avec l'exception `SignError`. Dans les deux cas, ceci laisse la possibilité à l'appelant la possibilité de décider ce qu'il va faire : laisser lui aussi l'exception se propager, rappeler la fonction avec une nouvelle chaîne, renvoyer une valeur par défaut, afficher un message...

Concernant le code, on utilise `int` pour transformer la chaîne en un entier. L'exception `ValueError` peut se produire si la chaîne ne correspond pas à un entier. Comme c'est avec ce type d'exception que `natural` doit signaler que ce problème, on laisse donc l'exception se propager. Si la chaîne correspond bien à un entier, on doit lever l'exception `SignError` si l'entier est strictement négatif. Dans le cas contraire, on a bien l'entier naturel attendu que l'on peut donc retourner.

Exercice 3 : Lecture robuste d'un entier naturel

Écrire (fichier `natural.py`) un sous-programme robuste, appelé `input_natural` qui réalise la saisie d'un entier naturel auprès d'un utilisateur. À l'image de `input`, l'appelant précisera la consigne à afficher à l'utilisateur. Si l'utilisateur ne saisit pas un entier, ce sous-programme affichera le message « Un entier est attendu ». Si un entier strictement négatif est saisi, le message « Un entier positif est attendu ». En cas de mauvaise saisie, l'utilisateur sera à nouveau sollicité.

On pourra la tester via un programme principal et le fichier `test_input_natural.py`.

Solution :

```
def input_natural(consigne):
    '''
    Réaliser une saisir robuste d'un entier naturel.
    La consigne est affichée avant chaque saisie.
    '''
    while True:
        try:
            return natural(input(consigne))
        except ValueError:
            print("Un entier est attendu.")
        except SignError:
            print("Un entier positif est attendu.")
```

On sollicite l'utilisateur via `input` et on utilise ensuite `natural` pour obtenir l'entier naturel correspond à la chaîne saisie. Si une exception se produit, c'est que transformation n'a pas pu être réalisée. Le type de l'exception nous indique le problème. Il suffit donc de récupérer les deux exceptions et, pour chacune, afficher le message adapté.

En cas d'exception, il faut à nouveau demander un entier naturel. On ajoute donc une boucle avant le **`try`**. La condition est **`True`**. Elle s'arrêtera si le **`return`** est exécuté et donc si `natural` a bien retourné un entier naturel et n'a pas propagé d'exception.

On pourrait avoir une boucle infini si l'utilisateur ne rentre jamais un entier naturel.

Une alternative à la boucle serait d'utiliser la récursivité et, donc dans les deux **`except`**, d'ajouter **`return`** `input_natural(consigne)`. Ici on pourrait être confronté à **`RecursionError`** si l'utilisateur s'obstine à ne pas saisir un entier naturel.

Exercice 4 : Lire un entier

On veut écrire une fonction robuste appelée `input_int` pour lire au clavier un entier. Elle prend en paramètre le message à afficher avant de solliciter l'utilisateur. Si aucun message n'est fourni, rien n'est affiché. On peut préciser les bornes dans lequel doit se situer l'entier grâce à deux options min et max. L'entier doit être entre ces deux bornes, bornes comprises.

Voici quelques exemples d'utilisation :

```
mois = input_int('Numéro de mois', min=1, max=12)
positif = input_int('Un entier positif', min=0)
negatif = input_int('Un entier strictement négatif', max=-1)
n = input_int('Un entier quelconque')
a = input_int()
erreur = input_int('impossible !', min=1, max=0)      # provoque ValueError
erreur = input_int("pas d'intérêt !", min=5, max=5)   # provoque ValueError
```

On ne veut pas que les appels suivants soient possibles :

```
positif = input_int('Mois', 0)
erreur = input_int('Mois', 1, 12)
```

4.1. Écrire la spécification de cette fonction (fichier `input_int.py`). On la testera en utilisant le programme de test fourni `test_signature_input_int.py`.

Solution :

```
def input_int(invite='', *, min=None, max=None):
    '''Demander à l'utilisateur un entier en précisant éventuellement une
    valeur minimum et maximum.

    Returns
        Un entier qui respectent les contraintes précisées (min et max)

    Raises
        ValueError si les contraintes précisées sont impossibles
    '''
```

4.2. Écrire l'implantation de cette fonction et la tester en utilisant `test_input_int.py`.

Solution :

```

def _check_is_None_or_int(valeur, texte):
    '''
    Raises
        ValueError si valeur n'est pas de type int
    '''
    if valeur is not None and not isinstance(valeur, int):
        raise ValueError(texte + ' doit être un entier')

def input_int(invite='', *, min=None, max=None):
    '''Demander à l'utilisateur un entier en précisant éventuellement une
    valeur minimum et maximum.

    Returns
        Un entier qui respectent les contraintes précisées (min et max)

    Raises
        ValueError si les contraintes précisées sont impossibles
    '''
    # Vérifier les paramètres nommés min et max
    _check_is_None_or_int(min, 'min')
    _check_is_None_or_int(max, 'max')
    if min is not None and max is not None:
        if min >= max:
            raise ValueError(f'Pas assez de valeurs possibles : '
                    'min={min} et max={max}')

    # Construire la chaîne expliquant les contraintes sur l'entier à saisir
```

```
    if min is None:
        if max is None:
            contraintes = ''
        else:
            contraintes = f' (au plus {max})'
    elif max is None:
        contraintes = f' (au moins {min})'
    else:
        contraintes = f' (entre {min} et {max})'

    # Définir le prompt
    if not invite:
        prompt = 'Un entier'
    else:
        prompt = invite
    prompt += contraintes + ' : '

    saisie_ok = False
    while not saisie_ok:
        try:
            # Saisir un entier
            reponse = input(prompt)
            entier = int(reponse)

            # Vérifier le respects des bornes (saisie_ok: out)
            if min is not None and entier < min:
                print('Cet entier est trop petit ! '
                      f'Il doit être supérieur ou égal à {min}.')
            elif max is not None and entier > max:
                print('Cet entier est trop grand ! '
                      f'Il doit être inférieur ou égal à {max}.')
            else:
                saisie_ok = True
        except ValueError:
            print('Un entier est attendu !')
    return entier
```

3 Fichiers

Exercice 5 : Fichiers et exceptions

La fonction `somme` du module `somme_reels.py` calcule la somme des nombres contenus dans le fichier dont le nom est passé en paramètre. Il doit y avoir un nombre par ligne.

```
import sys

def somme(nom_fichier: str) -> float:
    '''
```

```
    La somme des nombres qui sont dans le fichier dont le nom est en paramètre.
    Il doit y avoir un et un seul nombre par line.

    :param nom_fichier: le nom du fichier qui contient les nombres
    :return: la somme des nombres contenu dans le fichier, un par ligne
    '''
    with open(nom_fichier, 'r') as fichier:
        total = 0.0
        for ligne in fichier:
            nombre = float(ligne)
            total = total + nombre
        return total

def main():
    ''' Afficher la somme des réels d'un fichier. '''
    # Définir le nom du fichier
    if len(sys.argv) == 2: # via la ligne de commande
        nom = sys.argv[1]   # le nom du fichier
    else:                  # via l'utilisateur du programme
        nom = input('Nom du fichier : ').strip()

    print(somme(nom))

if __name__ == '__main__':
    main()
```

5.1. Exécuter ce programme sur le fichier `exemple1.txt` en faisant :

```
python somme_reels.py exemple1.txt
```

Que peut-on en conclure ?

Solution : Le fichier est :

```
10
5
4.5
-18
```

Le résultat est correct : 1.5.

Le programme a fonctionné comme attendu sur l'exemple 1.

5.2. Exécuter ce programme sur le fichier `exemple2.txt`. Que peut-on en conclure ?

Solution : Le fichier est :

```
2
5
xxx
a
6

-1
```

Le programme se termine sur une exception non récupérée :

```
ValueError: could not convert string to float: 'xxx\n'
```

Le programme n'est donc pas robuste quand le fichier est mal formé... Et on apprend que c'est l'exception `ValueError` qui est levée par `float()` si une ligne du fichier ne correspond pas à un réel.

5.3. Modifier le programme pour ignorer toutes les lignes du fichier qui ne sont pas des nombres. On récupèrera l'exception qui s'est produite lors de l'exécution précédente. Pour bien voir les lignes ignorées, on écrira un message « Ligne ignorée : » suivi du contenu de la ligne en question.

On affichera le numéro de ligne qui a été ignorée.

Solution : On peut donc récupérer l'exception et alors ignorer la donnée et afficher le message indiqué.

```python
def somme(nom_fichier: str) -> float:
    '''
    La somme des nombres qui sont dans le fichier dont le nom est en paramètre.
    Il doit y avoir un et un seul nombre par line.

    :param nom_fichier: le nom du fichier qui contient les nombres
    :return: la somme des nombres contenu dans le fichier, un par ligne
    '''
    with open(nom_fichier, 'r') as fichier:
        total = 0.0
        numero = 0      # Plus élégant : utiliser un enumerate dans le for
        for ligne in fichier:
            try:
                numero += 1
                nombre = float(ligne)
                total = total + nombre
            except ValueError:
                print(f'{nom_fichier}:{numero}: ligne ignorée `{ligne[:-1]}`')
        return total
```

Au lieu de gérer « à la main » le numéro de ligne, on peut s'appuyer sur la fonction `enumerate`. On obtient alors le code suivant.

```python
def somme(nom_fichier: str) -> float:
    '''
    La somme des nombres qui sont dans le fichier dont le nom est en paramètre.
    Il doit y avoir un et un seul nombre par line.

    :param nom_fichier: le nom du fichier qui contient les nombres
    :return: la somme des nombres contenu dans le fichier, un par ligne
    '''
    with open(nom_fichier, 'r') as fichier:
        total = 0.0
        for numero, ligne in enumerate(fichier, 1):
            try:
```

```
13                  nombre = float(ligne)
14                  total = total + nombre
15              except ValueError:
16                  print(f'{nom_fichier}:{numero}: ligne ignorée `{ligne[:-1]}`')
17          return total
```

Erreurs possibles : Le code suivant contient la plupart des erreurs possibles. Saurez-vous les identifier ?

```
1   def somme(nom_fichier: str) -> float:
2       '''...'''
3       with open(nom_fichier) as fichier:
4           numero = 0
5           total = 0.0
6           for ligne in fichier:
7               try:
8                  nombre = float(ligne)
9                  numero += 1
10              except ValueError:
11                  print(f'Ligne {numero} ignorée :', ligne, end='')
12              total = total + nombre
13          return total
```

5.4. Exécuter ce programme sur `exemple3.txt`, fichier qui n'existe pas. Que peut-on en conclure ?
Solution : Le programme se termine sur une exception, `FileNotFoundError`. Il n'est donc toujours pas robuste !

Cette exception est levée par la fonction `open` parce que le fichier `exemple3.txt` n'existe pas.

5.5. Modifier le programme pour qu'il soit plus robuste.
Solution : Il faut se poser la question de où récupérer cette exception.

Ce pourrait être dans `somme` mais que faire alors ? On ne sait pas d'où vient le nom du fichier. On ne sait donc pas ce qu'il faut faire pour avoir un nom de fichier valide. C'est certainement le sous-programme appelant qui sait quoi faire. On doit donc laisser se propager l'exception.

C'est donc dans l'autre fonction que l'on va récupérer l'exception. On peut afficher un message. On pourrait aussi (re)demander un nom du fichier à l'utilisateur. D'autres stratégies seraient possibles...

```
1   def main():
2       ''' Afficher la somme des réels d'un fichier. '''
3       # Définir le nom du fichier
4       if len(sys.argv) == 2: # via la ligne de commande
5           nom = sys.argv[1]   # le nom du fichier
6       else:                  # via l'utilisateur du programme
7           nom = input('Nom du fichier : ').strip()
8
9       try:
10          print(somme(nom))
11      except FileNotFoundError:
12          print(f"Le fichier {nom} n'existe pas.")
13      except Exception as exception:
14          print(exception)
```

Pour avoir un exemple d'exception qui n'est pas explicitement traité par le programme précédent, on peut essayer de lire un fichier pour lequel on ne possède pas les droits en lecture.

Sous Unix (en bash), on peut faire :

```
touch exemple4.txt      # créer un fichier exemple4.txt vide
chmod u-r exemple4.txt  # supprimer (-) le droit de lecture (r) à l'utilisateur (u)
python somme_reels.py exemple4.txt
```

5.6. Dans le fichier `somme_reels_tous.py`, compléter le code de la fonction `somme_tous` qui prend en paramètre une liste de nom de fichiers et retourne un triplet avec d'abord le cumul des sommes des fichiers qui ont pu être traités, la liste des fichiers qui n'existent pas et, enfin, la liste des fichiers qui ont provoqué un problème d'entrée sortie.

Solution : Dans cette fonction on récupère donc l'exception `FileNotFoundError` pour ajouter le nom dans une première liste et `IOError` pour ajouter le nom dans une autre liste.

```python
def somme_tous(noms_fichiers: list[str]) -> tuple[float, list[str], list[str]]:
    '''
    Retourne la somme des valeurs dans les fichiers.
    Les fichiers doivent contenir une seule valeur par ligne.
    Seuls les fichiers qui ont pu être lu correctement sont pris en compte.
    Les fichiers qui contiennent des erreurs sont listés ainsi que ceux qui
    contiennent n'existe pas ou provoque des erreurs lors de leur lecture.

    :param noms_fichiers: liste des noms des fichiers à traiter.
    :return: un triplet (cumul, fichiers_inexistants, fichiers_erreur) où
        - cumul : la somme des réels des fichiers qui ont pu être traités complètement
        - fichiers_inexistants : la liste des fichiers inexistants
        - fichiers_erreur : la liste des fichiers en erreur et donc ignorés
    '''
    fichiers_inexistants = []
    fichiers_erreurs = []
    cumul = 0.0
    for nom in noms_fichiers:
        try:
            cumul += somme(nom)
        except FileNotFoundError:
            fichiers_inexistants.append(nom)
        except:
            fichiers_erreurs.append(nom)
    return cumul, fichiers_inexistants, fichiers_erreurs
```