Exceptions

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')
```

1.2. Comment faire pour récupérer toutes les exceptions ?

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`).
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.

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`.

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

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

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.

5.4. Exécuter ce programme sur `exemple3.txt`, fichier qui n'existe pas. Que peut-on en conclure ?

5.5. Modifier le programme pour qu'il soit plus robuste.

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.