Gestion d'exceptions et tests unitaires

Avant-propos :

Cette feuille d'exercices a pour objectif d'introduire les notions de gestion d'exceptions et de test unitaire. Nous vous proposons de vous familiariser avec ces concepts par le biais de l'implémentation d'exemples simples en Python.

1 Gestion d'exceptions

1.1 Definition

Un programme écrit en Python est constitué d'un ensemble d'instructions qui sont exécutées les unes après les autres dans un ordre spécifique. Au cours de l'exécution, une erreur peut survenir et casser le flot instructions; on parle alors d'exception. Il existe de nombreuses causes possibles pour l'émission d'une exception:

- lorsqu'on tente d'accéder à une valeur/variable/index qui n'existe pas
- lorsqu'on tente d'appeler une opération qui n'est pas définie (nulle part/pour un type de donnée/pour une certaine valeur...)
- etc.

A savoir : Exceptions

Une exception (originellement "exceptional event" en anglais) est "quelque chose d'inattendu" qui se passe pendant l'exécution d'un programme et qui perturbe le flot de ses instructions. Quand un tel événement est reconnu, le programme rassemble de l'information concernant sa nature et son contexte dans un objet/-variables/etc (dépend du langage) appelé une "exception" et la propage en remontant la pile des appels (up the call stack), c'est-à-dire vers le scope englobant le programme. Pendant qu'elle se propage vers le haut, une exception peut être "attrapée" et traitée de sorte à ce que le programme puisse reprendre une exécution normale depuis un état "réparé". Si l'exception parvient à rejoindre le scope le plus haut, c'est-à-dire (dans la plupart des langages) celui du "main", alors (dans la plupart des langages), le programme se termine avec un code de retour non nul (signifiant une erreur).

Dans les langages orientés objet comme Python, Java ou C++, la gestion d'exception repose sur la manipulation d'objets de types dérivant d'une classe "Exception". On peut ensuite sélectivement les gérer selon leurs types.

En Python, lorsqu'une exception est levée sur une certaine ligne d'instruction, le programme s'interrompt à cette ligne. L'objet "exception" créé est accompagné d'un objet "traceback" qui permet de déterminer le parcours (suite d'appels de méthodes et lignes dans le code source) du flot d'instruction ayant mené à l'apparition de l'erreur.

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

Si l'exception n'est pas arrêtée, le programme (en mode script) s'arrête définitivement (en effet, il vaut mieux stopper le programme plutôt que de continuer avec un état et des structures de données corrompues); les informations recueillies sont alors imprimées dans la sortie d'erreur (*stderr*).

1.2 Usages

Toutefois on peut intercepter une exception qui est levée afin de ne pas faire terminer le programme, on parle alors de gestion d'exception. Pour ce faire on utilise en Python le motif try/catch:

```
try:
    instructionsBlock1
except ExceptionName:
    instructionsBlock2
```

Il vaut mieux alors traiter l'exception de manière à ne pas laisser le programme dans un état qui va immédiatement causer une seconde exception. Dans l'exemple ci-dessus, en supposant que "*instructionsBlock*1" est censé remplir un certain objectif, on peut, avec "*instructionsBlock*2":

- annuler l'objectif et réinitialiser les variables du programme
- essayer de remplir le même objectif avec un autre méthode
- logger l'erreur et continuer l'exécution
- etc.

Il existe de nombreux types d'exceptions fournis nativement par Python et ses librairies ("*IndexError*", "*ZeroDivisionError*","*AttributeError*",...). Ces exceptions sont levées automatiquement lorsque certaines conditions sont rencontrées pendant l'exécution du programme. En plus de cela, Python permet au programmeur de lever lui même des exceptions à l'aide du mot clef "*raise*":

```
if (condition) :
    instructionsBlock1
else:
    raise ExceptionName(args)    # depending on their type,
                                 # exceptions may accept different arguments
```

Dans la plupart des langages orientés objet, comme en Python, les exceptions sont toujours rattrapables.

On peut aussi définir nos propres exceptions en définissant des classes héritant des exceptions de base Python. Nous n'allons pas aborder en détail ces concepts de la programmation orientée objet mais simplement introduire une syntaxe pour définir des exceptions basiques:

```
class MyException (Exception): pass
```

Cette ligne déclare une exception "*MyException*" qui peut dès lors être levée ou rattrapée par le programme.

1.3 Exercices

Exercice 1 : Inverse d'un entier 😎

Dans cet exercice nous verrons comment intercepter et traiter des exceptions qui sont levées dans une partie du code que nous ignorons (ici, dans des méthodes de la librairie standard Python).

A savoir : Input

Il existe de nombreuses façons d'échanger de l'information avec un programme durant son exécution (interface réseau, interface textuelle ou graphique, etc.).
Nous utiliserons ici la fonction "input", qui permet au programme de recevoir des informations textuellement de la part de l'utilisateur pendant son exécution. Cela consiste à interrompre le programme, donner la main

à l'utilisateur pour une entrée clavier, et puis convertir en une variable "string" la chaîne de caractères entrée par l'utilisateur.

A savoir : Output

Il existe plusieurs manières d'"écrire" sur le terminal dans lequel on a lancé un programme Python. Ici, nous allons uniquement utiliser la très simple fonction "print".

Considérons le programme ci-dessous. La fonction "invert_a" calcule et imprime l'inverse d'un nombre entré par l'utilisateur. Comme vous pouvez le constater, aucun contrôle n'est fait sur les données entrées par l'utilisateur.

```
1 def invert_a():
2     my_number = input('please enter a number ')
3     my_number = int(my_number)
4     my_inverted = 1/my_number
5     print('the inverse of {} is {}'.format(my_number,my_inverted))
6
7 if __name__ == '__main__':
8     invert_a()
```

Question 1: Trouvez deux entrées possibles qui lèverons 2 exceptions différentes dans ce programme. Ces erreurs sont associées à des exceptions spécifiques dont vous noterez les noms (qu'il est nécessaire de connaître pour pouvoir les intercepter précisément).

Question 2: A partir de `invert_a`, écrire une nouvelle version `invert_b` de la fonction qui met en place des vérifications suffisantes pour que les 2 exceptions levées dans la questions précédente ne le soient plus. A la place, un message doit être affiché, avertissant l'utilisateur qu'il a entré des données erronées.

Question 3: A partir de `invert_a`, écrire une nouvelle version `invert_c` ayant le même comportement que `invert_b` mais faisant les vérifications *a posteriori* avec le motif try/except.

Exercice 2 : Terminal d'inscription automatique 😎

Dans cet exercice, nous:

- verrons comment définir nos propres exceptions
- pourrons observer comment fonctionne la propagation des exceptions

Nous donnons ci-dessous du code Python, qui, dans l'esprit, programme un terminal d'inscription automatique permettant de s'inscrire à divers évènements. Vous pouvez vite constater qu'il s'agit seulement d'une façade; il s'agit juste de demander des inputs à l'utilisateur et, en fonction de vérifications arbitraires, d'afficher certaines choses et/ou de redemander des inputs.

```
1  class IncorrectEmailAdress(Exception): pass
2
3  class IncorrectPhoneNumber(Exception): pass
4
5  def get_email_address():
6      em = input("please enter your email address : ")
7      if em.endswith(".com") and (len(em.split(sep="@"))==2):
8          return({ "email" : em })
9      else:
10         raise IncorrectEmailAdress
11
```

```
12 def get_phone_number():
13     pn = input("please enter your phone number : ")
14     if( len(pn) != 8):
15         raise IncorrectPhoneNumber
16     for chara in pn:
17         try:
18             int(chara)
19         except ValueError:
20             raise IncorrectPhoneNumber
21     return({ "phone" : pn})
22 
23 FLF21 = "FLF21"
24 BGOLF2018 = "BGOLF2018"
25 NATO = "NATO"
26 events = [FLF21,BGOLF2018,NATO]
27 lb_FLF21       = "21st International Conference on Frog Leg Fricassee"
28 lb_BGOLF2018   = "2018 Belgian Underwater Golf Tournament"
29 lb_NATO      = "NATO extraordinary general meeting on penguin philology"
30 labels = { FLF21 : lb_FLF21, BGOLF2018 : lb_BGOLF2018, NATO : lb_NATO }
31 required = {    FLF21 : [get_email_address,get_phone_number],
32                 BGOLF2018 : [get_phone_number],
33                 NATO : []}
34 
35 def treat_registration(ev_name):
36     reg_file = {}
37     for rqrm in required[ev_name]:
38         while(True):
39             try:
40                 reg_file.update(rqrm())
41             except IncorrectEmailAdress:
42                 print("wrong input")
43             else:
44                 break
45     return reg_file
46 
47 def welcome_screen():
48     print("Welcome to the automated registration terminal")
49     print("-- Which event do you wish to register to ? --")
50     for ev in events:
51         print(ev + " - " + labels[ev])
52     ev_name = input()
53     subscription_file = treat_registration(ev_name)
54     print("congratulations, you are registered to " + ev_name)
55     print("registration information : " + str(subscription_file))
56 
57 if __name__ == '__main__':
58     welcome_screen()
```

Question 1: Ce programme est intentionnellement mal écrit. Trouvez 2 erreurs différentes qu'il est possible de déclencher simplement en entrant certaines séquences d'inputs.

Question 2: Certaines "erreurs" peuvent être déclenchées, non pas par des valeurs introduites par l'utilisateur ou l'environnement, mais par des valeurs de paramétrisation du programme. Typiquement il s'agirait de bases de données, de fichiers de configuration ou de variables globales du programme. Ici nous les trouverons en tant que variables globales aux lignes 23 à 31. Trouvez-y 2 modifications d'une seule valeur permettant chacune de déclencher une erreur différente.

Question 3: Proposez une nouvelle version du programme qui implémente les fonctionnalités suivantes:

- prise en compte des erreurs et cas non traités liés aux données entrées par l'utilisateur:

  - en interceptant les exceptions pertinentes dans la fonction "*treat_registration*"
  - lorsque l'utilisateur entre le nom d'un évènement n'existant pas

- prise en compte des erreurs liées aux données de paramétrisation:
  - dans l'impression initiale de la liste d'évènements sur l'écran d'accueil
  - quand l'utilisateur tente de s'inscrire à un évènement mal paramétré; alors, le programme doit écrire sur le terminal un message spécifiant que la procédure d'inscription à cet évènement est pour le moment non-fonctionelle

Question 4: La fonction "*treat_registration*" permet de récupérer un dossier d'inscription en en remplissant un à un tous les champs demandés. Pour cela elle lance autant de fonctions que nécessaire, fonctions paramétrées et spécifiés dans la variable globale "*required*" pour chaque évènement. Dans le programme, nous avons 2 fonctions, permettant respectivement de récupérer une adresse email et un numéro de téléphone.

Toutefois cette architecture nous permet de rajouter n'importe quel type de donnés dans un dossier d'inscription sans modifier le code originel et simplement en faisant référence à une fonction qui permet d'en récupérer une instance. En effet, il suffit de rajouter dans la liste liée à un évènement donné dans "*required*" la fonction que l'on veut lancer. Au moment de s'inscrire à l'évènement concerné, "*treat_registration*" lancera alors cette fonction automatiquement et stockera la valeur retournée dans le dossier d'inscription nouvellement créé.

Ce type d'architecture permet de respecter le principe d'ouverture/fermeture (open-closed principle) en ingénierie logicielle; principe stipulant que tout programme se doit d'être ouvert à l'extension mais fermé à la modification, c'est-à-dire qu'il devrait permettre son comportement d'être étendu à de nouveaux cas sans avoir besoin de modifier son code existant.

Dans cette question, nous vous demandons d'opérer ce genre d'extension. Vous implémenterez une fonction "*get_pin_number*" permettant d'obtenir un champ "*pin*" (code de 5 chiffres) dans un dossier d'inscription et l'utiliserez pour compléter le dossier de l'évènement "NATO".