DIU Enseigner l'informatique au lycée

Notes de cours

Kim Nguyễn

kim.nguyen@lri.fr

Mise au points de programmes

Notions introduites

- Gestion des erreurs
- Annotations de types
- Tests de correction

Gestion des erreurs

Exceptions (rappels)

Une exception est un **objet** dérivant de la classe `Exception`. Le constructeur de la classe attend en argument une chaîne de caractère (le message). Le type de la sous-classe indique la nature du problème. Quelques classes d'exceptions

`ArithmeticError` : levée en cas de problème lors d'opérations sur les nombres. Sous-classes : `OverflowError`, `ZeroDivisionError`, ...

`AssertionError` : levée par `assert` e lorsque e s'évalue à `False`.

`TypeError` : levée lorsque les arguments d'une fonction ne sont pas du type attendu.

`ValueError` : levée lorsqu'un argument est du bon type mais d'une valeur incorrecte

`LookupError` : levée lors d'un accès erroné dans une collection. Sous-classes : `KeyError` (clé inexistante dans un dictionnaire), `IndexError` (indice de tableau invalide).

Levée d'exception

Lorsque l'on est dans une **situation exceptionnelle** (i.e. lorsque le programme ne peut plus avancer) on peut « lever une exception » au moyen de l'instruction `raise`.

```
def nieme_element(l, n):
"""renvoie le nième élément d'une liste chaînée"""
    if l is None or n < 0:
        raise IndexError("Indice invalide : " + str(n))
    elif n = 0:
      return l.valeur
    else
      return nieme_element(l.suivante, n)
```

Situation exceptionnelle ?

Une situation est **exceptionnelle** pour une portion de code si, à cet endroit là du programme, on n'a pas d'autre solution que **d'interrompre le programme**.

C'est le comportement de l'instruction `raise` e. Sans mesure préventive particulière, le programme s'interrompt. L'interpréteur Python affiche dans la console une **trace de pile**, i.e. la liste de toutes les fonctions en cours d'exécution et non encore terminées (avec les numéros de lignes, noms de fichiers etc. . .).

Rattrapage d'exception

Une exception levée peut être **rattrapée** par un gestionnaire d'exception. Un tel gestionnaire se définit en Python par la construction `try:/except:`

```
try:
    ... code ...
except:
    ... gestion ...
```

Ici, le `code` peut lever des exceptions. L'exécution s'arrête alors à l'instruction `raise` et toutes les instructions suivantes sont **ignorées**. L'exécution reprend alors dans le bloc `except`. Les instructions de `gestion` peut gérer la situation.

« Philosophie » des exceptions

Programmer avec des exceptions repose sur les concepts suivants :

- Si tout se passe normalement, le code client est naturel, sans tests inutiles.
- En cas d'erreur, cette dernière est signalée au code appelant, le plus à même à gérer l'erreur.

Attention cependant, cette approche n'est pas parfaite :

- Si personne ne rattrape l'exception alors le programme plante sommairement
- Si on rattrape tous les types d'exception sans discrimination, on peut masquer des problèmes (débugage difficile)

Gestion avancée

Le bloc `try/except` peut filtrer les excpetions qu'il rattrape selon leur type :

```
try:
  ... code ...
except IndexError:
    ... gestion accès invalide ...
except ValueError:
    ... gestion valeur invalide ...
```

Le code ci-dessus installe de gestionnaires pour deux erreurs particulières. Si le `code` lève une exception `TypeError` cette dernière sera propagée au dela de ce bloc. Si personne ne la rattrape, le programme « plantera ».

Il est considéré comme une bonne pratique de rattraper uniquement les exceptions auxquelles on s'attend.

Gestion avancée (suite)

Les multiples « branches » `except` sont testée tour à tour. La première ayant une exception **compatible** avec l'exception levée est utilisée.

```
try:
  nieme_element(l, -100)    #leve IndexError
except LookupError:
      print("Ici !")
except IndexError:
      print("Pas ici !")
```

Le code ci-dessus affiche `"Ici !"` (car `LookupError` est une super classe de `IndexError`).

Gestion avancée (suite)

Lorsque l'on gère des ressources du systèmes (fichier, connexions réseau, ...) on veut souvent exécuter du code de « netoyage » quelle que soit la façon dont on termine l'exécution d'une fonction.

```
def sauver_dans_fichier(f):
    try:
        while ...:
            ... ecriture dans le fichier f ...

        f.close()
    except:
        f.close()
```

Dans l'exemple ci-desuss, si une erreur se produit dans la boucle while, on veut tout de même fermer le fichier (pour ne pas perdre ce qu'on a écrit jusque là par exemple). Ici le code est court. Mais si le code est plus compliqué ?

Gestion avancée (suite)

On peut compléter le bloc `try/except` par une branche `finally`

```
def sauver_dans_fichier(f):
    try:
        while ...:
            ... ecriture dans le fichier f ...
    except:
        ...
    finally:
        f.close()
```

Le code écrit dans `finally` est **toujours** exécuté, quelle que soit la façon dont on quitte le bloc `try` :

- sans erreur
- avec une exception
- avec un `return` dans le `try` ou `except` ! Dans ce cas, le code du `finally` est quand même exécuté avant de quiter la fonction.

Alternative aux exceptions

Une autre solution (pour les fonctions ou méthodes) consiste à renvoyer une valeur spéciale. La valeur peut être :

- Du même type que le résultat (quand c'est possible)
- `None`
- Une « encapsulation » du résultat dans un type différent

Comme les exceptions, aucune de ces approches n'est parfaite. On les illustre avec leurs limites dans la suite.

Exemple

Considérons une fonction `dept_ville(v)` qui renvoie le nom du département où se trouve la ville `v`. On regarde comment la fonction doit se comporter si la ville n'existe pas.

Valeur invalide du bon type

La fonction peut choisir de renvoyer la chaîne vide. Le code client est alors très simple :

```
d = dept_ville("Foobar")
d = a.upper()   # nom de département en majuscule
... faire quelque chose avec d ...
```

Le code est **trop** simple. La suite du code client utilise un nom de département invalide (""). Une erreur peut se produire beaucoup plus loin à cause de ce nom invalide.

None

La fonction peut choisir de renvoyer `None`. Le code client est adapté

```
d = dept_ville("Foobar")
if d is None:
      ... gérer l'erreur ...
d = a.upper()   # nom de département en majuscule
... faire quelque chose avec d ...
```

Le code est robuste, mais il impose une contrainte au client. En effet, il faut gérer l'erreur immédiatement. Ce n'est peut être pas approprié. De plus si cette fonction est utilisée souvent le code de gestion d'erreur peut être dupliqué.

None (suite)

`None` n'est pas forcément adapté. Dans l'exemple des listes chaînées, renvoyer `None` pour la fonction `nieme_element` est ambigu. Est-ce qu'on a effectivement trouvé la $n^{\text{ième}}$ valeur (qui valait `None`) ou est-ce qu'on a eu une erreur ?

Encodage

On peut choisir de renvoyer `(True, v)` si le résultat est sans erreur et `(False, None)`.

- Similaire à `None` dans le cas de `dept_ville`. On ne propage pas silencieusement une chaîne invalide, on doit tester avant d'extraire la seconde composante.
- Mieux que `None` dans le cas de `nieme_element`, (`(False, None)` indique une erreur, `(True, None)` indique que `None` était dans la liste).
- Oblige toujours à traiter l'erreur juste après l'appel
- Plus lourd et non-idiomatique en Python

Avec des exceptions

- Le code client peut gérer l'erreur au plus près s'il le souhaite (comme `None`) :

```
try:
  d = dept\_ville(v)
except ValueError:
    ... gérer l'erreur ...

d.upper()
```

- Mais il peut aussi laisser l'exception remonter et la capturer à un autre endroit (par exemple, une application graphique peut rattraper toutes les exceptions et afficher une boite de dialogue avec un message)

Conclusion sur les exceptions

+ Idiomatiques en Python
+ Laissent plus de choix au code client
  - On peut oublier de les gérer (et le programme plante)
  - On peut les gérer de façon trop générique (et introduire des bugs)

Types

– « Il n'y a pas de types en Python ! »
– « Si, mais ils sont juste là pour faire joli ! »
– « C'est déjà ça ! »

Types en Python

En programmation, la notion de **type** désigne une classification des objets manipulés en fonction de leur nature. D'un langage à l'autre, ces informations sont plus ou moins présentes, mais on gagne toujours à ne pas les ignorer.

Chaque valeur manipulée par un programme Python est associée à un type, qui caractérise la nature de cette valeur.

Types en Python

valeur	type	description
`1`	`int`	nombres entiers
`3.14`	`float`	nombres décimaux
`True`	`bool`	booléens
`"abc"`	`str`	chaînes de caractères
`None`	`NoneType`	valeur indéfinie
`(1, 2)`	`tuple`	n-uplets
`[1, 2, 3]`	`list`	tableaux
`{1, 2, 3}`	`set`	ensembles
`{'a': 1, 'b': 2}`	`dict`	dictionnaires

À l'exception de `NoneType` tous les noms de type ci-dessus sont en fait des **classes**. On peut donc appeler `int()` (le constructeur de la classe `int`), `dict()`, ...

Types en Python

Un programme Python peut vérifier le type d'une valeur et lever une exception en cas de type incompatible :

```
def dept_ville(v):
  if type(v) is not str:
    raise TypeError("Chaîne de caractères attendue!")
    ...
```

Cette vérification de type est faite à **l'exécution**, on parle de typage **dynamique**.

Annotations de type

En Python (depuis 3.5) il est possible d'ajouter à des programmes des annotations de type. Les fonctions, les méthodes et les définitions de variables peuvent être annotées :

```
x : int = 42

def pgcd(x : int, y : int) -> int:
  """Calcule le PGCD de deux entiers"""
   ...
```

Attention, ce ne sont que des **annotations** dont l'interpréteur Python ne se sert pas (et en particulier qui ne sont pas vérifiées).

```
x : int = "WAT"    # aucun problème :'(
```

À quoi ça sert ?

- À **documenter** son code
- À réfléchir **en amont** (avant d'écrire le code)
- À penser à des tests **en amont** (avant d'écrire le code)
- À **vérifier** son code au moyen d'outils externes (`mypy`, `pyrigh`, l'éditeur VSCode de Microsoft, . . .)

Exemple avec la classe Liste

```
class Liste:
  def __init__(self) -> None:
      ...
  def longueur(self) -> int:
        ...
  def est_vide(self) -> bool:
        ...
  def nieme_element(self, n : int) -> object:
        ...
  def ajoute(self, x : object) -> None:
        ...
  def retire(self) -> object:
        ...
  def concat(self, l2 : Liste) -> Liste:
        ...
```

Exemple avec la classe `Liste` (suite)

- On rappelle qu'en Python, une fonction sans `return` explicite, ou avec un `return` simple (sans valeur) renvoie `None`.
- Le type `object` est le type au sommet de la relation d'héritage. Toutes les valeurs Python sont du type `object`.
- Il est redondant d'annoter le type de `self` car il est toujours du type de la classe dans laquelle on est.

Exemple d'activité pour les élèves

- Placer les élèves en binôme.
- Leur donner une fonction non typée et sa spécificaiton en français.
- Ils réfléchissent à deux aux **types** de la fonction et les écrivent
- Binôme 1 code la fonction, Binôme 2 écrit des tests sans connaître le code.
- Ils testent jusqu'à obtention d'une fonction correcte.

Tests de correction

Tester un programme

On sait déjà (programme de 1ère) tester des fonctions avec :

- utilisation de `assert`
- tests des valeurs limites des arguments d'une fonction

On illustre maintenant comment tester une fonction non triviale par rapport à sa spécification. On considère une fonction de

```
tri(t : list) -> None
```

qui trie un tableau d'entiers en place.

Un programme de test

```
def test(t):
    tri(t)
    for i in range(0, len(t) - 1):
        assert t[i] <= t[i+1]
```

On souhaite montrer que cette fonction de `test` n'est pas suffisante, pour tester la correction du `tri`.

Que teste ce programme ?

- toues les éléments présents dans `t` après le tri sont dans l'ordre

Est-ce suffisant ?

Que teste ce programme ?

- toues les éléments présents dans `t` après le tri sont dans l'ordre

Est-ce suffisant ?

```
def tri(t):
    t.clear() #supprime tous les éléments du tableau
```

Que teste ce programme ?

- toues les éléments présents dans `t` après le tri sont dans l'ordre

Est-ce suffisant ?

```
def tri(t):
    t.clear() #supprime tous les éléments du tableau
```

- Il faut vérifier que l'on n'a pas perdu (ou ajouté) d'éléments !

(cf. exercices)

Comment tester ?

Une fois que l'on a une bonne fonction de test, il faut s'intéresser aux valeurs de tests. Sur quoi tester ?

- `[]`
- `[1, 2]`
- ...

Ce n'est pas suffisant !

Génération de tests aléatoires

On fait appel au module `random` et à la fonction `randint` :

```
from random import randint

def tableau_aleatoire(n, a, b):
    return [ randint(a, b) for _ in range(n) ]

for n in range(100):
    test(tableau_aleatoire(n, 0, 0))
    test(tableau_aleatoire(n, - n // 4, n // 4))
    test(tableau_aleatoire(n, - 10 * n, 10 * n))
    test(tableau_aleatoire(n, 0, n // 4))
    test(tableau_aleatoire(n, 0, n * 10))
    test(tableau_aleatoire(n, -n // 4, 0))
    test(tableau_aleatoire(n, -n * 10, 0))
```

Conclusion sur les tests

- Plus les algorithmes sont compliqués, plus il faut tester
- Il faut bien tester **toute** la spécification
- Il faut « industrialiser » les tests : tests automatiques sur des jeux de données variés, générés aléatoirement, mais sans oublier les cas limites.