Introduction à la Programmation des Algorithmes

6.2. Python – Exceptions, itérables et générateurs

François Fleuret

https://fleuret.org/11x001/

Python permet de gérer les erreurs et plus généralement les comportements anormaux d'un programme à l'aide **d'exceptions**.

Une opération qui provoque une exception provoque l'arrêt du programme si elle n'est pas gérée:

```
1  n = int("14")
2  print(n)
3
4  n = int("bateau")
5  print(n)
```

affiche

```
14
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    n = int("bateau")
ValueError: invalid literal for int() with base 10: 'bateau'
```

Une exception est une instance (un objet) d'une classe d'exceptions correspondant à un type d'erreur (erreur de conversion, division par zéro, fichier inexistant, etc.).

Quand un problème se produit une instance de la classe correspondante est créée et le flot normal du programme est interrompu.

Certaines de ces exceptions contiennent des informations relatives au problème rencontré (identifiant inconnu, type problématique, etc.)


```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam * 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Quand une exception se produit, l'interpréteur Python donne des informations sur la séquence d'appels de fonctions qui l'a causée.

```
1  def inverse(x):
2      return 1 / x
3
4  def truc(x):
5      return x * inverse(1 + x)
6
7  print(truc(3))
8  print(truc(-1))
9  print('Tout se finit bien')
```

affiche

```
0.75
Traceback (most recent call last):
  File "test.py", line 8, in <module>
    print(truc(-1))
  File "test.py", line 5, in truc
    return x * inverse(1 + x)
  File "test.py", line 2, in inverse
    return 1 / x
ZeroDivisionError: division by zero
```


Il est possible d'exécuter un bout de programme en indiquant quoi faire si une exception se produit à l'aide de `try` et `except`.

La partie de programme entre le `try` et le `except` est exécutée, et si une exception se produit et peut être traitée par le `except` elle l'est, ou bien le programme continue à être exécuté comme s'il n'y avait pas eu de `try`.

La gestion des exceptions se fait donc avec des constructions du type:

```
try:
    <clause possiblement problématique>

except TypeDErreur1:
    <clause qui traite l'exception>
except TypeDErreur2:
    <clause qui traite l'exception>
    /.../
else:
    <clause s'il n'y a pas eu d'exception>
```



```
try:
    n = int("bateau")
    print('Ça a marché!')

except ValueError:
    n = -1
    print('Oups, on gère l\'exception')

print(n)

print('Tout se finit bien')
```

affiche

```
Oups, on gère l'exception
-1
Tout se finit bien
```

```python
try:
    for k in range(5, -5, -1):
        print(f'1/{k} = {1 / k}')

except ZeroDivisionError:
    print('On essayé de diviser par zéro')

print('Tout se finit bien')
```

affiche

```
1/5 = 0.2
1/4 = 0.25
1/3 = 0.3333333333333333
1/2 = 0.5
1/1 = 1.0
On essayé de diviser par zéro
Tout se finit bien
```


Si le `except` ne correspond pas, l'exception n'est pas interceptée, et l'exécution continue comme s'il n'y avait pas eu de `try`.

```python
try:
    for k in range(5, -5, -1):
        print(f'1/{k} = {1 / k}')

except ValueError:
    print('Une valeur n\'est pas correcte')

print('Tout se finit bien')
```

affiche

```
1/5 = 0.2
1/4 = 0.25
1/3 = 0.3333333333333333
1/2 = 0.5
1/1 = 1.0
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    print(f'1/{k} = {1 / k}')
ZeroDivisionError: division by zero
```

Le `try` capture également les exceptions levées dans les fonctions appelées dans sa clause.

```
1  def inverse(x):
2      return 1 / x
3
4  def truc(x):
5      return x * inverse(1 + x)
6
7  try:
8      print(truc(3))
9      print(truc(-1))
10
11 except ZeroDivisionError:
12     print('On essayé de diviser par zéro')
13
14 print('Tout se finit bien')
```

affiche

```
0.75
On essayé de diviser par zéro
Tout se finit bien
```


On peut avoir plusieurs `except`.

```
1  def inverse(x):
2      return 1 / x
3
4  def truc(x):
5      return x * inverse(1 + x)
6
7  try:
8      print(truc(3))
9      print(truc(-1))
10
11 except ValueError:
12     print('Une valeur ne convient pas')
13 except ZeroDivisionError:
14     print('On a essayé de diviser par zéro')
15
16 print('Tout se finit bien')
```

affiche

```
0.75
On a essayé de diviser par zéro
Tout se finit bien
```

On peut rajouter un clause `else`.

```
1  def chose(x):
2      try:
3          r = 1 / x
4      except ZeroDivisionError:
5          r = 0
6          print('On essayé de diviser par zéro')
7      else:
8          print('Tout s\'est bien passé')
9
10     return r
11
12 print(chose(10))
13 print(chose(0))
14 print('Tout se finit bien')
```

affiche

```
Tout s'est bien passé
0.1
On essayé de diviser par zéro
0
Tout se finit bien
```


On peut imbriquer les `try`

```
1  try:
2      try:
3          r = 1 / x
4      except ZeroDivisionError:
5          print('On essayé de diviser par zéro')
6      except ValueError:
7          print('Une valeur est incorrecte')
8  except NameError:
9      print('Un identifiant est incorrect')
```

affiche

```
Un identifiant est incorrect
```

Il est possible de récupérer des informations relative à l'exception en associant l'instance de l'exception à une variable avec le mot clé `as`

```
1  y = 5
2
3  try:
4      r = 1 / x
5
6  except NameError as ne:
7      print(f'Un identifiant est incorrect ({ne})')
```

affiche

```
Un identifiant est incorrect (name 'x' is not defined)
```

Une exception peut avoir des variables d'instance qui lui son propres: nom de fichier, identifiant, numéro d'erreur, etc.

On peut lancer des exceptions avec `raise`

```
1  def affiche_etoiles(n):
2      if type(n) != int or n <= 0:
3          raise ValueError(f'la valeur {n} n\'est pas un entier positif')
4      print('*' * n)
5
6  affiche_etoiles(8)
7  affiche_etoiles(1.2)
```

affiche

```
********
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    affiche_etoiles(1.2)
  File "test.py", line 3, in affiche_etoiles
    raise ValueError(f'la valeur {n} n\'est pas un entier positif')
ValueError: la valeur 1.2 n'est pas un entier positif
```

On peut créer ses propres exceptions mais cela sort du cadre de ce cours.

Cela demande en particulier d'utiliser des mécanismes d'héritage qui sont propres à la programmation orientée objets.

Itérables et itérateurs

Python permet de parcourir facilement les éléments des structures de données composées telles que les séquences:

```
1  for i in [ 0, -1, 2, -3 ]:
2      print(i)
```

En réalité, la boucle `for` se fait sur un **itérable**, qui est un objet capable de retourner un **itérateur**.

Un **itérateur** est un objet qui possède une méthode `__next__()` qui retourne "le prochaine élément" et doit provoquer une exception `StopIteration` s'il n'y en a plus.

Cette méthode peut être appelée explicitement ou bien via la fonction `next`.

```
1  class MonIterateur:
2      def __init__(self, p, m):
3          self.p = p
4          self.m = m
5          self.v = 1
6
7      def __next__(self):
8          if self.v >= self.m:
9              raise StopIteration()
10         else:
11             v = self.v
12             self.v *= self.p
13             return v
```

```
i = MonIterateur(3, 100)

print(next(i))
print(next(i))
print(next(i))
print(next(i))
print(next(i))
print(next(i))
```

affiche

```
1
3
9
27
81
Traceback (most recent call last):
  File "test.py", line 28, in <module>
    print(next(i))
  File "test.py", line 15, in __next__
    raise StopIteration()
StopIteration
```


Un **itérable** est un objet qui possède une méthode `__iter__` qui retourne un **itérateur**.

```
class MonIterable:
    def __init__(self, p, m):
        self.p = p
        self.m = m

    def __iter__(self):
        return MonIterateur(self.p, self.m)

for k in MonIterable(3, 100):
    print(k)
```

affiche

```
1
3
9
27
81
```

On peut construire une séquence à partir d'un itérable:

```
1  print(list(MonIterable(2, 20)))
2
3  print(tuple(MonIterable(10, 100000)))
```

affiche

```
[1, 2, 4, 8, 16]
(1, 10, 100, 1000, 10000)
```


Il est possible de définir un itérable qui est aussi un itérateur. Cela simplifie puisqu'il faut une seule classe, mais fait que cet itérable ne pourra être parcouru qu'une seule fois.

```
1  class Cross:
2      def __init__(self, n):
3          self.n = n
4          self.v = 0
5          self.d = 1
6
7      def __iter__(self):
8          return self
9
10     def __next__(self):
11         if self.v >= 0 and self.v < self.n:
12             r = '-' * self.v + '+' + '-' * (self.n - self.v - 1)
13             self.v += self.d
14             return r
15         else:
16             raise StopIteration()
```

```
for s in Cross(5):
    print(s)
```

affiche

```
+----
-+---
--+--
---+-
----+
```


Et un itérateur peut avoir une méthode `__reversed__` qui retourne un itérateur qui itère les mêmes valeurs en sens inverse:

```
    def __reversed__(self):
        self.d = -1
        self.v = self.n - 1 - self.v
        return self

for s in reversed(Cross(8)):
    print(s)
```

affiche

```
-------+
------+-
-----+--
----+---
---+----
--+-----
-+------
+-------
```

Il est possible de combiner plusieurs itérables en un seul itérable sur des tuples avec `zip`:

```
>>> for k in zip(range(3), [ 'a', 'b', 'c' ]): print(k)
...
(0, 'a')
(1, 'b')
(2, 'c')
>>> for i, j in zip(range(3), [ 'a', 'b', 'c' ]): print(i, j)
...
0 a
1 b
2 c
```

Ce nouvel itérateur a autant de termes que celui qui en a le moins.


```
for s, t, n in zip(Cross(4),
                   reversed(Cross(4)),
                   range(0, 8, 2)):
    print(s, t, n)
```

affiche

```
+--- ---+ 0
-+-- --+- 2
--+- -+-- 4
---+ +--- 6
```

Finalement, `enumerate` permet de construire un itérable sur des paires avec un indexe numérique:

```
>>> for i, x in enumerate([ 'un', 'deux', 'trois' ]): print(i, x)
...
0 un
1 deux
2 trois
```


Générateurs

Python offre une autre manière de faire des itérateurs à l'aide de l'instruction `yield`. Cette instruction s'utilise dans le corps d'une fonction et permet de spécifier la séquence de valeurs à renvoyer.

Une fonction avec un `yield` retourne un **générateur** qui possède un `__next__` et `__iter__`.

Chaque appelle à `__next__` de ce générateur continue l'exécution de la fonction jusqu'au prochain `yield` et retourne sa valeur.


```
1  def fruits():
2      yield 'pomme'
3      yield 'poire'
4      yield 'fraise'
5
6  g = fruits()
7
8  print(next(g))
9  print(next(g))
10 print(next(g))
11 print(next(g))
```

affiche

```
pomme
poire
fraise
Traceback (most recent call last):
  File "test.py", line 12, in <module>
    print(next(g))
StopIteration
```

```python
def fruits():
    yield 'pomme'
    yield 'poire'
    yield 'fraise'

for n in fruits():
    print(n)
```

affiche

```
pomme
poire
fraise
```



```python
def truc():
    yield 'attention'
    yield 'depart'
    for a in range(5):
        if a % 2 == 0:
            yield a
        else:
            yield -a

for n in truc():
    print(n)
```

affiche

```
attention
depart
0
-1
2
-3
4
```