

# Améliorer la qualité du code

# Table des matières

<b>I. Exceptions</b>	<b>3</b>
A. Exceptions.....	3
B. Assertions .....	4
<b>II. Exercice : Quiz</b>	<b>5</b>
<b>III. SOLID</b>	<b>6</b>
<b>IV. Exercice : Quiz</b>	<b>9</b>
<b>V. Essentiel</b>	<b>9</b>
<b>VI. Auto-évaluation</b>	<b>10</b>
A. Exercice .....	10
B. Test.....	10
<b>Solutions des exercices</b>	<b>12</b>

# I. Exceptions

Durée : 1 h

Environnement de travail : PyCharm

## Contexte

Quand vous obtenez une erreur ou une exception dans votre programme, cela signifie que tout le programme va arrêter de fonctionner. Vous ne voulez pas que cela se produise dans vos programmes. Il est préférable que le programme détecte les erreurs, les gère puis continue à fonctionner.

Un code épuré est constitué de peu d'instructions inutiles, peu d'erreurs et est généralement mieux structuré. Il est donc plus facile de le maintenir, avec des économies à la clé. Toutefois, dans la vraie vie, peu de développeurs débutants ont la chance de livrer un code conçu dans les règles de l'art. Même un développeur professionnel peut faire des erreurs à des moments donnés. En réalité, développer un logiciel est un processus complexe.

Il nécessite la participation de nombreux développeurs. Un code source peut de ce fait être travaillé et modifié plusieurs fois au cours du cycle de vie d'un logiciel. Il peut alors y avoir des lignes de code mal écrites ainsi que des redondances et des problèmes de nommage. Tous ces facteurs peuvent constituer des entraves au bon fonctionnement du logiciel et dégrader sa qualité et sa performance. Les bugs deviennent plus délicats à corriger et chaque ajout de fonctionnalité prend plus de temps. Il est donc important d'améliorer la qualité du code pour régler cette situation.

## A. Exceptions

Le mécanisme de gestion des exceptions est au cœur de Python. Ainsi, il n'existe pas de moyen de se retrouver avec une erreur qui ne soit pas une exception. Les exceptions se produisent lors de l'exécution.

Vous allez apprendre à lever ces exceptions plutôt que de laisser du code s'exécuter de manière incontrôlée, ce qui va créer des exceptions non maîtrisées par la suite. Il est préférable de prendre les devants, de prévoir les désagréments que l'on souhaite éviter afin de générer soi-même une exception adaptée à la situation.

## Définition

### Qu'est-ce qu'une exception en Python ?

Une exception est, en Python, une instance d'une classe héritée de « *BaseException* ». En effet, la clause « *except* » dans un bloc « *try* » traite de la classe d'exception et également des autres classes dérivées de cette classe. Notez que 2 classes qui ne sont pas liées par héritage ne peuvent en aucun cas être équivalentes.

Il est possible de lever les exceptions natives en utilisant l'interpréteur ou les fonctions natives. En effet, une valeur associée peut indiquer la cause de l'erreur. Il peut s'agir d'une chaîne ou d'un n-uplet qui contient de nombreux éléments d'erreurs.

## Les erreurs

Nous avons défini une fonction appelée `spam`, en lui attribuant un paramètre. En imprimant la valeur de cette fonction avec ces paramètres, une erreur `ZeroDivisionError` se produit chaque fois que vous essayez de diviser un nombre par zéro.

À partir du numéro de ligne dans le message d'erreur, vous savez que la déclaration `return` dans `spam()` provoque une erreur. Les erreurs peuvent être gérées avec les instructions `try` et `except`.

Le code qui pourrait potentiellement générer une erreur est mis dans une clause `try`. L'exécution du programme se déplace au début de la clause `except` suivante en cas d'erreur. Vous pouvez placer le code de division par zéro précédent dans une clause `try` et avoir une clause `except` contenant du code pour gérer l'erreur.

Lorsque le code dans une clause `try` provoque une erreur, l'exécution du programme passe immédiatement au code de la clause `except`. Après avoir exécuté le code, l'exécution se poursuit normalement.

Notez que toutes les erreurs qui se produisent dans les appels de fonction dans un bloc `try` seront également identifiées. La raison pour laquelle l'impression (`spam(1)`) n'est jamais exécutée est qu'une fois que le code se déplace dans la clause `except`, il ne revient pas à la clause `try`. Au lieu de cela, il continue de s'exécuter normalement.

#### Exemple

```
1 < def spam(divideBy) :
2 return 42 / divideBy
3 print(spam(2))
4 print(spam(12))
5 print(spam(0))
6 print(spam(1)) >

1 < def spam(divideBy):
2 try:
3 return 42 / divideBy
4 except ZeroDivisionError:
5 print('Erreur, l'argument est invalide')
6 print(spam(2))
7 print(spam(12))
8 print(spam(0))
9 print(spam(1)) >
```

## B. Assertions

### Définition Qu'est-ce qu'une assertion en Python ?

On définit les assertions comme étant un outil permettant de détecter les erreurs dans les programmes Python. Ils permettent de rendre ces programmes fiables et faciles à déboguer.

### Fondamental Le rôle des assertions en Python

`Assert` est une instruction de Python qui permet de corriger les exceptions et qui teste une condition. Ainsi, votre programme va fonctionner normalement une fois que la condition est vraie. Cependant, lorsque la condition d'assertion est erronée, l'exception `AssertionError` est alors levée avec un message d'erreur facultatif.

Pour bien utiliser les assertions, il faut rapidement informer les développeurs des erreurs irrécupérables qui se trouvent dans un programme. Les assertions ne permettent pas de signaler les conditions d'erreur attendues à l'instar d'un « *fichier introuvable* ». En fait, les conditions d'erreurs attendues peuvent être facilement corrigées par un utilisateur.

Par ailleurs, on peut dire que les assertions constituent des auto-contrôles internes pour votre programme. Mais comment fonctionnent-elles ?

Elles évoluent de sorte à déclarer certaines conditions comme impossibles dans votre code. En effet, si toutes ces conditions ne sont pas remplies, on peut dire qu'il y a un bogue dans le programme. Notez que si votre programme ne contient pas de bogue, aucune de ces conditions ne se produira. Si l'une de ces conditions se produit, le programme arrêtera de fonctionner.

Cela vous révèle quelle assertion a produit l'erreur. Ainsi, vous pouvez déterminer quelle condition impossible a été déclenchée. En utilisant les assertions, vous pouvez donc aisément faire la recherche et la correction de bogues dans vos programmes.

### Rôle de l'instruction d'assertion

L'instruction d'assertion est utile pour permettre de générer des exceptions si des conditions ne sont pas requises dans un objectif de contrôle de la qualité. Les assertions peuvent être utilisées simplement pour tester une expression.

Il ne se passe alors rien tant que tout va bien. Une exception de type `AssertionError` est produite si l'expression se révèle incorrecte. Il est alors possible de passer au mot-clé une seconde expression qui est évaluée et restituée en cas de problème et qui permet de préciser un peu mieux le type de problème rencontré.

#### Exemple

```
1 < a=1
2 assert a%2 == 1
3 a=2 >

1 < a=1
2 assert a%2 ==1, 'variable a non correcte'
3 a=2
4 assert a%2 ==1, 'variable a non correcte' >
```

### Exercice : Quiz

[solution n°1 p.13]

#### Question 1

En Python, toutes les erreurs sont des exceptions.

- ☐ Vrai
- ☐ Faux

#### Question 2

Pour lever les exceptions, il est possible de faire usage de l'interpréteur ou des fonctions natives.

- ☐ Vrai
- ☐ Faux

#### Question 3

Une assertion permet de détecter les erreurs dans les programmes Python.

- ☐ Vrai
- ☐ Faux

#### Question 4

Les conditions d'erreurs attendues peuvent être signalées par les assertions.

- ☐ Vrai
- ☐ Faux

#### Question 5

Les bogues contenus dans un code source peuvent être détectés par les assertions.

- ☐ Vrai
- ☐ Faux

### III. SOLID

#### Définition La notion de SOLID

SOLID est une abréviation qui regroupe un ensemble de principes de conception créés pour le développement de logiciels dans des langages orientés objet. Les principes de SOLID visent à favoriser un code plus simple, plus robuste et actualisable. Chaque lettre en SOLID correspond à un principe de développement.

Lorsqu'il est mis en œuvre correctement, il rend votre code plus logique et plus facile à lire.

#### Le SOLID

- **Single Responsibility Principle - Principe de responsabilité unique**

Le principe de responsabilité unique exige qu'une classe n'ait qu'une seule fonction. Si une classe a plus d'une responsabilité, elle devient couplée. Le changement de responsabilité entraîne la modification de l'autre responsabilité.

Nous avons une classe `User` qui est responsable à la fois des propriétés de l'utilisateur et de la gestion de la base de données des utilisateurs. Si l'application change d'une manière qui affecte les fonctions de gestion de la base de données, les classes qui utilisent les propriétés d'utilisateur devront être touchées et recompilées pour compenser les nouvelles modifications. C'est un effet domino, touchez un pion, cela affecte tous les autres.

Nous divisons simplement la classe, nous créons une autre classe qui stocke un utilisateur dans une base de données.

- **Open-Closed principle - Principe ouvert-fermé**

Les entités logicielles (classes, modules, fonctions) doivent être ouvertes à l'extension et non à la modification.

Imaginons que vous ayez un magasin et que vous accordiez une réduction de 20 % à vos clients préférés en utilisant cette classe. Cela ne respecte pas le principe d'OCP. Si vous voulez donner un nouveau pourcentage de remise à un autre type de clients, vous pouvez le faire différemment. En suivant le principe d'OCP, nous ajouterons une nouvelle classe qui prolongera la remise. Dans cette nouvelle classe, nous implémenterons son nouveau comportement.

- **Liskov Substitution Principle - Principe de substitution de Liskov**

Si `S` est un sous-type de `T`, alors les objets de type `T` peuvent être remplacés par des objets de type `S` sans altérer le fonctionnement du programme.

Nous allons voir des exemples de code en Python qui violent ce principe. Commençons par un code valide qui contient une calculatrice. La fonction de calcul renvoie toujours un nombre (le produit de `a` et `b`).

Chaque sous-classe doit implémenter une fonction de calcul qui renvoie un nombre. Brisons le principe en créant une fonction de calcul qui peut également générer une erreur. Cet exemple ajoute une classe `DividerCalculator` (héritée de `Calculator`) où la fonction de calcul surchargée génère une erreur lorsque Python tente de diviser par zéro.

Il n'y a aucun moyen de corriger ce code sans :

- Refactoriser la hiérarchie des classes.
- Enveloppez chaque appel de calcul avec `try / except`.

La classe `DividerCalculator` est différente de la classe `Calculator` :

- La multiplication de 2 nombres donne toujours un nombre.
- La division de 2 nombres entraîne un nombre ou une erreur.

Multiplier et diviser sont des actions différentes avec un résultat différent, ils ne doivent donc pas être dérivés l'un de l'autre.

### • Interface Segregation Principle - Principe de séparation des interfaces

Ce principe traite des inconvénients de la mise en œuvre d'interfaces complexes. Les utilisateurs ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas.

Une seule classe peut implémenter plusieurs interfaces. Nous pouvons fournir une seule implémentation pour toutes les méthodes communes entre les interfaces.

Les interfaces séparées vous obligent également à penser davantage à votre code du point de vue de l'utilisateur. Non seulement vous améliorez votre code pour nos utilisateurs, mais vous le rendez également plus facile à comprendre, à tester et à mettre en œuvre.

### • Dependency Inversion Principle - Principe d'inversion de dépendance

- Les modules de haut niveau ne doivent pas dépendre de modules de bas niveau.
- Les classes de bas et de haut niveau doivent dépendre des mêmes abstractions.
- Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre d'abstractions.

Il arrive un moment dans le développement logiciel où votre application sera en grande partie composée de modules. Lorsque cela arrive, nous devons clarifier les choses avec l'injection de dépendances. Pour créer un comportement spécifique, vous pouvez utiliser des techniques telles que l'héritage ou les interfaces.

#### Exemple

1 < #ci-dessous la une classe a deux responsabilités

```
2 class User:
3     def __init__(self, name: str):
4         self.name = name
5     def get_name(self) -> str:
6         pass
7     def save(self, user: User):
8         pass >
```

```
1 < class User:
2     def __init__(self, name: str):
3         self.name = name
4     def get_name(self):
5         pass
6 class UserDB:
7     def get_user(self, id) -> User:
8         pass
9     def save(self, user: User):
10        pass >
```

```
1 < class Discount:
2     def __init__(self, customer, price):
3         self.customer = customer
4         self.price = price
5     def give_discount(self):
6         if self.customer == 'fav':
7             return self.price * 0.2
8         if self.customer == 'vip':
9             return self.price * 0.4 >
```

```
1 < class Discount:
2     def __init__(self, customer, price):
3         self.customer = customer
4         self.price = price
5     def get_discount(self):
6         return self.price * 0.2
```

```

7 class VIPDiscount(Discount):
8     def get_discount(self):
9         return super().get_discount() * 2 >
1 < def get_discount(self) :
2     return super().get_discount() *2 >
1 < class Calculator():
2     def calculate(self, a, b): # returns a number
3         return a * b
4
5 calculation_results = [
6     Calculator().calculate(3, 4),
7     Calculator().calculate(5, 7),
8 ]
9 print(calculation_results) >
1 < class Calculator():
2     def calculate(self, a, b): # returns a number
3         return a * b
4 class DividerCalculator(Calculator):
5     def calculate(self, a, b): # returns a number or raises an Error
6         return a / b
7 calculation_results = [
8     Calculator().calculate(3, 4),
9     Calculator().calculate(5, 7),
10    DividerCalculator().calculate(3, 4),
11    DividerCalculator().calculate(5, 0) # 0 will cause an Error
12 ]
13 print(calculation_results) >
1 < class IShape:
2     def draw(self):
3         raise NotImplementedError
4 class Circle(IShape):
5     def draw(self):
6         pass
7 class Square(IShape):
8     def draw(self):
9         pass
10 class Rectangle(IShape):
11     def draw(self):
12         pass >
1 < class AuthenticationForUser():
2     def __init__(self, connector:Connector):
3         self.connection = connector.connect()
4     def authenticate(self, credentials):
5         pass
6     def is_authenticated(self):
7         pass
8     def last_login(self):
9         pass
10 class AnonymousAuth(AuthenticationForUser):
11     pass
12 class GithubAuth(AuthenticationForUser):
13     def last_login(self):
14         pass
15 class FacebookAuth(AuthenticationForUser):
16     pass

```



```
17 class Permissions()  
18     def __init__(self, auth: AuthenticationForUser)  
19         self.auth = auth  
20     def has_permissions():  
21         pass  
22 class IsLoggedInPermissions (Permissions):  
23     def last_login():  
24         return auth.last_log >
```

## Exercice : Quiz

[solution n°2 p.13]

### Question 1

SOLID fait allusion à un ensemble de principes de conception conçu pour le développement de logiciels.

- ☐ Vrai
- ☐ Faux

### Question 2

Les 5 lettres du SOLID correspondent à un principe de développement.

- ☐ Vrai
- ☐ Faux

### Question 3

Selon le principe de responsabilité, une classe peut avoir plusieurs fonctions.

- ☐ Vrai
- ☐ Faux

### Question 4

Le principe ouvert-fermé stipule que les entités logicielles telles que les modules, les classes et les fonctions peuvent être ouverts à la modification.

- ☐ Vrai
- ☐ Faux

### Question 5

Selon le principe d'inversion de dépendance, les détails dépendent des abstractions.

- ☐ Vrai
- ☐ Faux

## V. Essentiel

Nous retenons qu'une exception est une instance de classe provenant de la `BaseException` en Python. En fait, les erreurs issues de l'exécution d'un programme Python se présentent sous forme d'exception. Ainsi, une exception est constituée des données sur le contexte de l'erreur. Lorsqu'elle n'est pas traitée, elle peut entraîner l'interruption du programme. Pour lever les exceptions, on utilise les mots-clés « *try* » et « *except* ». Notez que *except* a la capacité de traiter non seulement l'exception qu'elle mentionne, mais aussi toutes les classes dérivées de cette classe.

Par ailleurs, pour détecter les exceptions, on utilise les assertions. Grâce à l'instruction `assert` de Python, vous pouvez corriger les bogues. Cependant, les assertions ne gèrent pas les erreurs d'exécution. Leur rôle est de permettre aux développeurs d'identifier la cause d'un bogue.

Il est indispensable d'aborder la notion de SOLID. Cet acronyme résume les 5 premiers principes de la conception orientée objet. En outre, ces principes forment aussi une philosophie de base pour des méthodologies comme le développement agile et le développement de logiciels adaptatifs.

## VI. Auto-évaluation

### A. Exercice

Vous allez créer une calculatrice interactive. En donnée d'entrée, l'utilisateur est supposé renseigner une formule composée d'un nombre, d'un opérateur (+ ou -) et d'un autre nombre, séparés par un espace (par exemple 1 + 1).

#### Question 1

[solution n°3 p.14]

Divisez l'entrée à l'aide de `str.split()` et vérifiez si le résultat est valide :

- Si l'entrée ne se compose pas des 3 éléments, générez une `FormulaError` avec une exception personnalisée.
- Essayez de convertir la première et la troisième entrée en un `float` (comme ceci : `float_value = float(str_value)`). Interceptez n'importe quelle `ValueError` qui se produit et déclenchez à la place une `FormulaError`.
- Si la deuxième entrée n'est pas '+' ou '-', déclenchez à nouveau une `FormulaError`.

Si les données d'entrées sont valides, effectuez le calcul et imprimez le résultat. L'utilisateur est alors invité à fournir une nouvelle entrée, et ainsi de suite, jusqu'à ce que l'utilisateur quitte le programme.

Une interaction avec le programme pourrait ressembler à ceci :

```
1 >>> 1 + 1
2 2.0
3 >>> 3.2 - 1.5
4 1.7000000000000002
5 >>> quit >
```

Vous venez d'intégrer une équipe de développement. Votre première tâche est de vous assurer que le dernier projet respecte les principes SOLID.

#### Question 2

[solution n°4 p.15]

Supposons que la classe ci-dessous est utilisée à la fois pour effectuer certaines opérations de base de données et pour accéder aux attributs de classe. Proposez une solution qui gère les propriétés de `Post` pour se conformer aux principes.

```
1 class Post:
2     def __init__(self, body):
3         self.body = body
4     def get_body(self):
5         pass
6     def save(self, post):
7         pass
```

### B. Test

#### Exercice 1 : Quiz

[solution n°5 p.15]

Question 1

Quand est-ce que le bloc `else` d'une exception `try-except-else` est exécutée ?

- ☐ Toujours
- ☐ Lorsqu'une exception se produit
- ☐ Lorsqu'aucune exception ne se produit
- ☐ Lorsqu'une exception se produit dans le bloc d'exception

#### Question 2

Le code suivant est valide.

```
1 < try:
2     # Action 1
3 except:
4     # Action 2
5 else:
6     # Action 3 >
```

- ☐ Faux, à cause du bloc `else`
- ☐ Faux, `else` ne peut pas être utilisé avec `except`
- ☐ Faux, `else` doit être avant `except`
- ☐ Vrai

#### Question 3

Un bloc d'instructions `except` peut gérer plusieurs exceptions.

- ☐ Vrai, à l'exception de `TypeError`, `SyntaxError` [...]
- ☐ Vrai, à l'exception `[TypeError, SyntaxError]`
- ☐ Faux
- ☐ Aucun des éléments mentionnés

#### Question 4

Quel bloc vous permet de tester un bloc de code pour les erreurs ?

- ☐ Try
- ☐ Except
- ☐ Except
- ☐ Aucun

#### Question 5

Quel sera le résultat du code suivant ?

```
1 < try:
2     print(x)
3 except:
4     print("Erreur") >
```

- ☐  $\leq x$
- ☐ Erreur
- ☐ Error
- ☐ Aucun


## Solutions des exercices

**Exercice p. 5 Solution n°1****Question 1**

En Python, toutes les erreurs sont des exceptions.

☒ Vrai

☐ Faux


 Le mécanisme de gestion des exceptions est au cœur de Python. Donc, il n'existe pas de moyen de se retrouver avec une erreur qui ne soit pas une exception.

**Question 2**

Pour lever les exceptions, il est possible de faire usage de l'interpréteur ou des fonctions natives.

☒ Vrai

☐ Faux


 Vous pouvez lever les exceptions natives en utilisant l'interpréteur ou les fonctions natives.

**Question 3**

Une assertion permet de détecter les erreurs dans les programmes Python.

☒ Vrai

☐ Faux


 En utilisant les assertions, vous pouvez facilement détecter les bugs dans les programmes Python.

**Question 4**

Les conditions d'erreurs attendues peuvent être signalées par les assertions.

☐ Vrai

☒ Faux


 Les conditions d'erreurs attendues peuvent être aisément corrigées par un utilisateur. Les assertions ne s'occupent pas donc de ce type d'erreur.

**Question 5**

Les bogues contenus dans un code source peuvent être détectés par les assertions.

☒ Vrai

☐ Faux

 Les assertions peuvent être utilisées pour tester une expression. Rien ne se passe lorsque tout est normal dans votre code source. Mais lorsqu'il y a une erreur, les expressions se produisent.


**Exercice p. 9 Solution n°2**

### Question 1

SOLID fait allusion à un ensemble de principes de conception conçu pour le développement de logiciels.

☒ Vrai

☐ Faux


 SOLID est un acronyme regroupant cinq principes de conception créés pour le développement de logiciels dans des langages orientés objet.

### Question 2

Les 5 lettres du SOLID correspondent à un principe de développement.

☒ Vrai

☐ Faux


 Le SOLID vise à favoriser un code plus simple, plus robuste et actualisable. Ainsi, les 5 lettres constitutives de l'acronyme SOLID correspondent chacune à un principe de développement.

### Question 3

Selon le principe de responsabilité, une classe peut avoir plusieurs fonctions.

☐ Vrai

☒ Faux


 Selon le principe de responsabilité unique, une classe n'a droit qu'à une seule fonction.

### Question 4

Le principe ouvert-fermé stipule que les entités logicielles telles que les modules, les classes et les fonctions peuvent être ouverts à la modification.

☐ Vrai

☒ Faux


 Les entités logicielles (classes, modules, fonctions) doivent être ouvertes à l'extension et non à la modification selon le principe ouvert-fermé.

### Question 5

Selon le principe d'inversion de dépendance, les détails dépendent des abstractions.

☐ Vrai

☒ Faux

 Le principe d'inversion de dépendance stipule que les abstractions ne doivent pas dépendre des détails. Toutefois, les détails doivent dépendre d'abstractions.

```

1 class FormulaError(Exception): pass
2 def parse_input(user_input):
3     input_list = user_input.split()
4     if len(input_list) != 3:
5         raise FormulaError('Input does not consist of three elements')
6     n1, op, n2 = input_list
7     try:
8         n1 = float(n1)
9         n2 = float(n2)
10    except ValueError:
11        raise FormulaError('The first and third input value must be numbers')
12    return n1, op, n2
13 def calculate(n1, op, n2):
14     if op == '+':
15         return n1 + n2
16     if op == '-':
17         return n1 - n2
18     if op == '*':
19         return n1 * n2
20     if op == '/':
21         return n1 / n2
22     raise FormulaError('{0} n'est pas un opérateur valide'.format(op))
23 while True:
24     user_input = input('>>> ')
25     if user_input == 'quit':
26         break
27     n1, op, n2 = parse_input(user_input)
28     result = calculate(n1, op, n2)
29     print(result)

```

#### p. 10 Solution n°4

```

1 class Post:
2     def __init__(self, body):
3         self.body = body
4     def get_body(self):
5         pass

```

```

1 class PostDB:
2     def save(self, post):
3         pass

```

#### Exercice p. 10 Solution n°5

##### Question 1

Quand est-ce que le bloc `else` d'une exception `try-except-else` est exécutée ?

- ☐ Toujours
- ☐ Lorsqu'une exception se produit
- ☒ Lorsqu'aucune exception ne se produit
- ☐ Lorsqu'une exception se produit dans le bloc d'exception

Q Else signifie sinon donc si y a une exception on fait telle chose sinon on fait une autre.

## Question 2

Le code suivant est valide.

```
1 < try:
2     # Action 1
3 except:
4     # Action 2
5 else:
6     # Action 3 >
```

- ☐ Faux, à cause du bloc else
- ☐ Faux, else ne peut pas être utilisé avec except
- ☐ Faux, else doit être avant except
- ☒ Vrai

Q Faire action 1 si exception faire action 2 sinon faire action 3.

## Question 3

Un bloc d'instructions `except` peut gérer plusieurs exceptions.

- ☒ Vrai, à l'exception de `TypeError`, `SyntaxError` [...]
- ☐ Vrai, à l'exception [`TypeError`, `SyntaxError`]
- ☐ Faux
- ☐ Aucun des éléments mentionnés

Q Un bloc `except` traite de la classe d'exception et également des autres classes dérivées de cette classe.

## Question 4

Quel bloc vous permet de tester un bloc de code pour les erreurs ?

- ☒ Try
- ☐ Except
- ☐ Except
- ☐ Aucun

Q On utilise le bloc Try pour gérer toutes les erreurs.


## Question 5

Quel sera le résultat du code suivant ?

```
1 < try:
2     print(x)
3 except:
4     print("Erreur") >
```



- ☐  $\leq x$
- ☒ Erreur
- ☐ Error
- ☐ Aucun

 S'il il y a une erreur écrire X sinon écrire erreur.