**UTILISER DES FONCTIONS RÉCURSIVES EN PYTHON** [**https://www.pythoniste.fr/python/utiliser-des-fonctions-recursives-en-python/**](https://www.pythoniste.fr/python/utiliser-des-fonctions-recursives-en-python/)

**Notre première fonction récursive**

Une fonction récursive est une fonction qui s’appelle elle-même jusqu’à ce qu’elle ne le fasse plus.

La fonction fn() suivante est une fonction récursive car elle a un appel à elle-même :

def fn():

# ...

fn()

# ...

Pour pouvoir s’arrêter, une fonction récursive doit avoir une condition d’arrêt. Nous devons donc ajouter une instruction if comme celle-ci :

def fn():

# ...

if condition:

# ne s'appelle pas elle-même

else:

fn()

# ...

En règle générale, on utilise une fonction récursive pour diviser un problème difficile à résoudre en problèmes plus petits qui sont plus faciles à résoudre.

En programmation, vous trouverez souvent les fonctions récursives utilisées dans les structures de données et les algorithmes comme les arbres, les graphiques et les recherches binaires.

**Exemples de fonctions récursives Python**

Prenons quelques exemples d’utilisation des fonctions récursives **Python**.

**Un exemple de fonction récursive simple en Python**

Supposons que nous ayons besoin de développer une fonction de *compte à rebours* qui compte à rebours à partir d’un nombre spécifié jusqu’à zéro.

Par exemple, si vous appelez la fonction qui compte à rebours à partir de 3, elle affichera la sortie suivante :

3

2

1

Ce qui suit définit la fonction count\_down() :

def count\_down(nombre\_de\_depart):

""" Compte à rebours à partir d'un nombre """

print(nombre\_de\_depart)

Si vous appelez la fonction count\_down() maintenant :

count\_down(3)

Elle n’affichera que le numéro de départ : 3.

Pour afficher respectivement dans l’ordre les nombres 3, 2 et 1, nous devons :

*Tout d’abord, appeler count\_down(3) pour afficher 3.  
Puis, appeler count\_down(2) pour afficher 2.  
Enfin, appeler count\_down(1) pour afficher 1.  
Pour ce faire, à l’intérieur de la fonction count\_down(), nous devons définir une logique pour appeler la fonction count\_down() avec les arguments 2 et 1.*

*Pour ce faire, nous devons rendre la fonction count\_down() récursive.*

Ce qui suit définit une fonction count\_down() récursive et l’appelle en passant le nombre 3 :

def count\_down(nombre):

""" Compte à rebours à partir d'un nombre """

print(nombre)

count\_down(nombre-1)

count\_down(3)

Si nous exécutons le programme, nous verrons l’erreur suivante :

RecursionError: maximum recursion depth exceeded while calling a Python object

*La raison en est que count\_down()s’appelle indéfiniment jusqu’à ce que le système l’arrête. Par défaut,* ***Python*** *considère que la limite maximale à ne pas dépasser est de 1000 appels.  
Nous pouvons vérifier cette limite à l’aide de* sys.getrecursionlimit() :

>>>import sys

>>>

>>>print(sys.getrecursionlimit())

1000

Et au besoin nous pouvons le modifier avec sys.setrecursionlimit() :

>>>import sys

>>>

>>>sys.setrecursionlimit(1500)

>>>print(sys.getrecursionlimit())

1500

Attention, si cette limite existe c’est qu’il y a une bonne raison.

Revenons en à notre compte à rebours, nous devons l’arrêter une fois le nombre zéro atteint. Pour ce faire, ajoutons une condition comme celle-ci :

def count\_down(nombre):

""" Compte à rebours à partir d'un nombre """

print(nombre)

# Appeler la fonction count\_down si le nombre suivant à décompter est positif

nombre\_suivant = nombre - 1

if nombre\_suivant > 0:

count\_down(nombre\_suivant)

count\_down(3)

Sortie:

3

2

1

Dans cet exemple, la fonction count\_down() s’appelle uniquement lorsque le nombre suivant qui doit être décompté est supérieur à zéro. En d’autres termes, si sa valeur est zéro, la fonction arrête de s’appeler.

**Utiliser une fonction récursive pour calculer la somme d’une séquence**

Supposons que nous ayons besoin de calculer la somme d’une séquence, par exemple de 1 à 100. Un moyen simple de le faire est d’utiliser une boucle for avec la fonction range() :

def somme(n):

total = 0

for index in range(n+1):

total += index

return total

result = somme(100)

print(result)

Sortie :

5050

Pour faire la même chose en utilisant la récursivité, vous pouvez calculer la somme de la séquence de 1 à n comme suit :

* *somme(n) = n + somme(n-1)*
* *somme(n-1) = n-1 + somme(n-2)*
* *…*
* *somme(0) = 0*

La fonction somme() continuera de s’appeler tant que son argument sera supérieur à zéro.

Ce qui suit définit la version récursive de la fonction somme() :

def somme(n):

if n > 0:

return n + somme(n-1)

return 0

result = somme(100)

print(result)

Comme vous pouvez le voir, la fonction récursive est beaucoup plus courte et plus lisible.

Si vous utilisez l’opérateur ternaire, le code de la fonction somme() sera encore plus concis :

def somme(n):

return n + somme(n-1) if n > 0 else 0

result = somme(100)

print(result)

Ce qu’il faut retenir de la récursivité :

* *Une fonction récursive est une fonction qui s’appelle elle-même jusqu’à ce qu’elle ne le fasse plus ;*
* *Une fonction récursive doit toujours avoir une condition d’arrêt ;*
* *Privilégier une approche récursive pour les problèmes qui peuvent être décomposés en plusieurs problèmes plus petits ;*
* *Mettre en cache des données avec @lru\_cache quand c’est possible ;*
* *Ne pas dépasser la taille de la pile.*

**L’INSTRUCTION D’ASSERTION EN PYTHON** [**https://www.pythoniste.fr/python/linstruction-dassertion-en-python-assert/**](https://www.pythoniste.fr/python/linstruction-dassertion-en-python-assert/)

Les assertions sont un outil très pratique qui aide à détecter automatiquement les erreurs dans vos programmes Python afin de les rendre plus fiables et plus faciles à déboguer.

**Que sont les assertions et à quoi servent-elles ?**

L’instruction assert de **Python** est une aide au débogage qui teste une condition. Si la condition est vraie, cela ne fait rien et votre programme continue simplement à s’exécuter. Mais si la condition d’assertion est fausse, elle lève une exception AssertionError avec un message d’erreur facultatif.

L’utilisation appropriée des assertions est d’**informer les développeurs d’erreurs irrécupérables** dans un programme. **Elles ne sont pas destinées à signaler les conditions d’erreur attendues** tel qu’un « fichier introuvable », où un utilisateur peut prendre des mesures correctives pour palier au problème puis simplement réessayer d’appeler le programme.

Une autre façon de voir les choses est de dire que les assertions sont des auto-contrôles internes pour votre programme. Elles fonctionnent en déclarant certaines conditions comme impossibles dans votre code. Si l’une de ces conditions n’est pas remplie, cela signifie qu’il y a un bogue (**bug**) dans le programme.

Si votre programme est exempt de bogues, ces conditions ne se produiront jamais. Mais si l’un d’entre-eux se produit, alors le programme plantera en vous révélant qu’elle assertion a généré une erreur, ce qui vous permettra de comprendre exactement quelle condition « impossible » a été déclenchée. Cela rend beaucoup plus facile la recherche et la correction des bogues (bugs) dans vos programmes.

Pour résumer : l’instruction assert de **Python** est une **aide au débogage (debug). Elle n’est pas un mécanisme de gestion des erreurs d’exécution.** L’objectif de l’utilisation des assertions est de permettre aux développeurs de trouver plus rapidement la cause probable d’un bogue. Une erreur d’assertion ne devrait jamais être déclenchée à moins qu’il n’y ait un bogue dans votre programme.

**Exemple d’Assertion en Python**

Commençons par un exemple simple afin que vous puissiez voir où les assertions peuvent être utiles. J’ai essayé de donner à mon exemple un semblant de problème qui pourrait nous arriver dans le monde réel et que vous pourriez rencontrer dans l’un de vos programmes.

Supposons que vous construisiez une boutique en ligne avec **Python**. Vous travaillez à ajouter une fonctionnalité de coupon de réduction au système et éventuellement à écrire la fonction appliquer\_reduction(prix, reduction) suivante :

*def appliquer\_reduction(prix, reduction):*

*prix\_reduit = prix - ( prix \* reduction )*

*assert 0 <= prix\_reduit <= prix*

*return prix\_reduit*

Avez-vous remarqué la déclaration assert dans la fonction ? Sa présence garantira que, quoi qu'il en soit, les prix réduits ne peuvent pas être inférieurs à 0 et ils ne peuvent pas être supérieurs au prix d'origine du produit.

Assurons-nous que cela fonctionne réellement comme prévu si nous appelons cette fonction pour appliquer une remise valide :

***# Appliquer une réduction de 25% sur un prix de 100 € -> 75 €***

*>>> appliquer\_reduction(100.00, 0.25)*

*75.0*

D'accord, cela a bien fonctionné. Essayons maintenant d'appliquer des remises non valides :

***# Appliquer une remise de 200 %***

*>>> appliquer\_reduction(100.00, 2.0)*

*Traceback (most recent call last):*

*File "<stdin>", line 1, in <module>*

*File "<stdin>", line 3, in appliquer\_reduction*

*AssertionError*

***# Appliquer une remise de -30 %***

*>>> appliquer\_reduction(100.00, -0.30)*

*Traceback (most recent call last):*

*File "<stdin>", line 1, in <module>*

*File "<stdin>", line 3, in appliquer\_reduction*

*AssertionError*

Comme vous pouvez le voir, essayer d'appliquer une remise non valide lève une exception AssertionError qui signale la ligne avec la condition d'assertion violée. Si jamais nous rencontrons l'une de ces erreurs lors du test de notre boutique en ligne, il sera facile de découvrir ce qui s'est passé en examinant les logs.

C'est le pouvoir des assertions, en un mot.

**La syntaxe d'assert**

C'est toujours une bonne idée d'étudier comment une fonctionnalité de langage est réellement implémentée dans **Python** avant de commencer à l'utiliser. Jetons donc un coup d'œil rapide à la syntaxe de l'instruction assert selon la documentation **Python** :

assert\_stmt ::= "assert" expression1 ["," expression2]

expression1 correspond à la condition que nous testons, et l'optionnel expression2 est un message d'erreur qui ne s'affichera que si l'assertion échoue.

Au moment de l'exécution, l'interpréteur **Python** transforme chaque instruction assert en ce qui suit :

*if \_\_debug\_\_:*

*if not expression1:*

*raise AssertionError(expression2)*

Vous pouvez utiliser expression2 pour transmettre un message d'erreur facultatif qui sera affiché avec l'erreur AssertionError dans le traceback. Cela peut simplifier encore plus le débogage. Par exemple, j'ai déjà vu du code comme celui-ci :

*if condition == 'x':*

*do\_x()*

*elif condition == 'y':*

*do\_y()*

*else:*

*assert False, ("Cela ne devrait jamais arriver, mais cela arrive de temps en temps."*

*"Nous essayons actuellement de comprendre pourquoi."*

*"Envoyez un e-mail à pythoniste.fr si vous rencontrez ce message.")*

Est-ce que c'est moche ? Carrément ! Mais c'est certainement une technique valable et utile si vous êtes confronté à un problème de type heisenbug (bogue difficile à localiser parce qu'il ne se manifeste pas lorsque des outils de détection sont utilisés pour le rechercher) dans l'une de vos applications.

**Pièges courants avec l'utilisation d'assertions en Python**

Avant de continuer, il y a deux mises en garde importantes que j'aimerai vous communiquer concernant l'utilisation d'assertions en **Python**.

La première concerne l'introduction de risques de sécurité et de bogues dans vos applications, et la seconde concerne une bizarrerie de syntaxe qui facilite l'écriture d' assertions inutiles.

**Mise en garde n°1 – N'utilisez pas d'assertions pour la validation des données**

Les assertions peuvent être désactivées globalement dans l'interpréteur **Python**. Ne comptez donc pas sur l'exécution des expressions d'assertion pour la validation ou le traitement des données.

Les assertions peuvent être globalement désactivées en ligne de commande et aussi via une variable d'environnement dans **CPython**.

Voici un exemple en utilisant le flag -O (**Optimize**) qui désactive toutes les assertions :

$ python -Oc "assert False"

$ python -c "assert False"

Traceback (most recent call last):

File "<string>", line 1, in <module>

AssertionError

La désactivation des assertions transforme toute instruction d'assertion en une opération nulle : les assertions sont simplement compilées et ne seront pas évaluées, ce qui signifie qu'aucune des expressions conditionnelles ne sera exécutée.

Il s'agit d'une décision de conception intentionnelle utilisée de la même manière par de nombreux autres langages de programmation. Comme effet secondaire, il devient extrêmement dangereux d'utiliser des instructions assert comme moyen rapide et facile de valider des données d'entrée.

Si votre programme utilise des assertions pour vérifier si un argument de fonction contient une valeur inattendue, cela peut rapidement se retourner contre vous et entraîner des bogues ou des failles de sécurité.

Jetons un coup d'œil à un exemple simple en reprenant notre application de boutique en ligne avec **Python**. Quelque part dans votre code d'application, il existe une fonction pour supprimer un produit à la demande d'un utilisateur :

def supprimer\_produit(produit\_id, utilisateur):

assert utilisateur.is\_admin(), 'Doit avoir des privilèges d'administrateur pour supprimer'

assert boutique.produit\_exist(produit\_id), 'Id produit inconnu'

boutique.trouver\_produit(produit\_id).delete()

Regardez attentivement cette fonction. Que se passe-t-il si les assertions sont désactivées ?

Il y aura deux problèmes sérieux dans cet exemple de fonction à trois lignes, causés par l'utilisation incorrecte des instructions assert :

* La vérification des privilèges d'administrateur avec une instruction assert est dangereuse. Si les assertions sont désactivées dans l'interpréteur **Python**, cela se transforme en une opération nulle. Par conséquent, tout utilisateur peut désormais supprimer des produits . La vérification des privilèges ne s'exécute même pas. Cela introduit probablement un problème de sécurité et ouvre la porte aux attaquants pour détruire ou endommager gravement les données de la boutique en ligne de votre client ou de votre entreprise. Pas bon ;
* La vérification produit\_exist() est ignorée lorsque les assertions sont désactivées. A partir de là, trouver\_produit() peut désormais être appelé avec des identifiants de produit invalides, ce qui pourrait entraîner des bogues plus graves selon la façon dont notre programme est écrit. Dans le pire des cas, cela pourrait être un moyen pour quelqu'un de lancer des attaques par déni de service contre notre magasin. par exemple, si l'application du magasin se bloque lorsque nous tentons de supprimer un produit inconnu, il est possible qu'un attaquant le bombarde de demandes de suppression non valides et provoque une panne.

Comment pouvons-nous éviter ces problèmes? La réponse est de **ne pas utiliser d'assertions pour valider les données**. Au lieu de cela, nous pourrions effectuer notre validation avec des instructions if régulières et **lever des exceptions de validation** si nécessaire comme dans l'exemple si-dessous :

def supprimer\_produit(produit\_id, utilisateur):

if not utilisateur.is\_admin():

raise AuthError('Doit avoir des privilèges d'administrateur pour supprimer')

if not boutique.produit\_exist(produit\_id):

raise ValueError('Id produit inconnu')

boutique.trouver\_produit(produit\_id).delete()

Cet exemple mis à jour présente également l'avantage qu'au lieu de lever des exceptions AssertionError non spécifiques, il soulève désormais des exceptions sémantiquement correctes comme ValueError ou AuthError (que nous devrons définir nous-mêmes).

**Mise en garde n° 2 – Des assertions qui n'échouent jamais**

Il est facile d'écrire accidentellement des déclarations d'assertion **Python** qui sont toujours évaluées à True.

Par exemple, lorsque vous passez un tuple comme premier argument dans une instruction assert, l'assertion est toujours évaluée comme vraie et n'échoue donc jamais. Ainsi, cette assertion n'échouera jamais :

assert(1 == 2, 'Fail !')

Cela a à voir avec le fait que les tuples non vides sont toujours vrais en **Python**. Si vous passez un tuple à une instruction assert, la condition assert sera donc toujours vraie, ce qui à son tour rend l'instruction assert ci-dessus inutile car elle ne peut jamais échouer et déclencher une exception.

Il est relativement facile d'écrire accidentellement de mauvaises assertions multi-lignes en raison de ce comportement peu intuitif. Cela conduit rapidement à des cas de tests dysfonctionnels qui donnent un faux sentiment de sécurité dans notre code de test. Imaginez que vous ayez cette assertion quelque part dans votre suite de tests unitaires :

*assert (*

*counter == 10,*

*'Tous les élements ont été comptés.'*

*)*

Lors de la première inspection, ce cas de test semble tout à fait correct. Cependant, il ne détectera jamais un résultat incorrect car il sera toujours évalué à True, quel que soit l'état de la variable de counter.

Donc soyez vigilant avec l'utilisation de l'instruction assert car c'est assez facile de se tirer une balle dans le pied avec ça. Heureusement, il existe des contre-mesures que vous pouvez appliquer pour éviter que cette bizarrerie de syntaxe ne cause des problèmes, vérifier que vous avez bien activés les messages de Warning ! Si oui, vous obtiendrez cette information explicite :

>>> assert(1 == 2, 'Fail !')

:1: SyntaxWarning: assertion is always true, perhaps remove parentheses?

**Le mot de la fin**

Malgré ces mises en garde, je pense que les assertions de **Python** sont un outil de débogage puissant qui est souvent sous-utilisé par les développeurs **Python**.

Comprendre comment fonctionnent les assertions et quand les appliquer peut vous aider à écrire des programmes **Python** plus faciles à gérer et à déboguer. C'est une grande compétence à apprendre qui aidera à amener votre **Python** au niveau supérieur et fera de vous un bon Pythoniste.

Évidemment, pour aller plus loin avec l'instruction assert, la prochaine étape serait d'étudier l'incontournable pytest qui est un indispensable dans l'univers des tests. D'ailleurs, cette librairie fera l'objet d'un article dédié.

**What is Python Identifier?**

“An **identifier** is a name given to an entity”.

In very simple words, an identifier is a user-defined name to represent the basic building blocks of Python. It can be a variable, a function, a class, a module, or any other object.

Now you know what exactly identifiers are. So, how do we use them? We can’t use anything, there are some certain rules to keep in mind that we must follow while naming identifiers.

1. The Python identifier is made with a combination of lowercase or uppercase letters, digits or an underscore.

These are the valid characters. Lowercase letters (a to z), Uppercase letters (A to Z) ,Digits (0 to 9), Underscore (\_)

Examples of a valid identifier: num1, FLAG, get\_user\_name, userDetails, \_1234

2. An identifier cannot start with a digit. If we create an identifier that starts with a digit then we will get a syntax error.

3. We also cannot use special symbols in the identifiers name. Symbols like ( !, @, #, $, %, . ) are invalid.

4. A keyword cannot be used as an identifier. In Python, keywords are the reserved names that are built-in in Python.

They have a special meaning and we cannot use them as identifier names.

5. The length of the identifiers can be as long as you want. Of course, it can not be greater than the available memory, however, the PEP-8 standards rule suggests not to exceed 79 characters in a line.

**Testing the Validity of Python Identifiers**

Python has some helper functions that are useful when you are not sure whether a **string** is a **keyword** or a **valid identifier**.

1. To check whether a **string** is a **keyword** or not, we have a keyword module.

*import keyword*

*print( keyword.iskeyword(“var”) )*

*print( keyword.iskeyword(“False”) )*

*print( keyword.iskeyword(“continue”) )*

*print( keyword.iskeyword(“count”) )*

1. The **str.isidentifier()** function is used to check the **validity** of an identifier.

*print( “name”.isidentifier() ) true*

*print( “#today”.isidentifier() ) false*

*print( “\_12hello”.isidentifier() ) true*

*print( “8cellos”.isidentifier() ) false*

Following the naming conventions are mandatory for everyone. But that’s not it!

The Python community has made a few more **guidelines** that are not compulsory but it is advised to follow some practices that are better for everyone in understanding things. Let’s see what these guidelines are.

1. Class names should start with a **capital** **letter** and all the other identifiers should start with a **lowercase letter**.

2. Begin private identifiers with an **underscore (\_)**. Note, this is not needed to make the **variable private**. It is only for the ease of the programmer to easily distinguish between **private variables** and **public variables**.

3. Use **double underscores (\_\_)** around the names of magic methods and don’t use them anywhere else. Python built-in magic methods already use this notation. **For example:** \_\_init\_\_ , \_\_len\_\_ .

4. Double underscores are used only when you are dealing with **mangling** in Python.

5. Always prefer using names longer than one character. **index=1** is better than **i=1**

6. To combine words in an **identifier**, you should use **underscore(\_)**. **For example:** get\_user\_details.

7. Use camel case for naming the variables. **For example:** fullName, getAddress, testModeOn, etc.

**Reserved Classes of Python Identifiers**

Some classes in Python have special meanings and to identify them, we use patterns of leading and trailing underscores.

**1. Single leading underscore (\_\*)**

This identifier is used to store the result of the last evaluation in the interactive interpreter. These results are stored in the \_\_builtin\_\_ module. These are private variables and they are **not imported** by **“from module import \*”**

**2. Double leading and trailing underscores (\_\_\*\_\_)**

Only the **system-defined names** use this notation. They are defined by the interpreter and its implementations. It is not recommended to define additional names using this convention.

**3. Leading double underscores (\_\_\*)**

**Class-private name mangling:** These category names are used within the context of a class definition. They are re-written to use a mangled form and avoid name **clashes** between private variables of base and derived classes.

**INTRODUCTION TO THE PYTHON CUSTOM EXCEPTION**

<https://www.pythontutorial.net/python-oop/python-custom-exception/>

To create a custom [exception](https://www.pythontutorial.net/python-oop/python-exceptions/) class, you [define a class](https://www.pythontutorial.net/python-oop/python-class/) that [inherits](https://www.pythontutorial.net/python-oop/python-inheritance/) from the built-in Exception class or one of its subclasses such as ValueError class:

The following example defines a CustomException class that inherits from the Exception class:

*class CustomException(Exception):*

*""" my custom exception class """*

To raise the CustomException, you use the [raise](https://www.pythontutorial.net/python-oop/python-raise-exception/) statement. For example, the following uses the raise statement to raise the CustomException:

*class CustomException(Exception):*

*""" my custom exception class """*

*try:*

*raise CustomException('This is my custom exception')*

*except CustomException as ex:*

*print(ex)*

Like standard exception classes, custom exceptions are also classes. Hence, you can add functionality to the custom exception classes like:

* Adding attributes and [properties](https://www.pythontutorial.net/python-oop/python-properties/).
* Adding [methods](https://www.pythontutorial.net/python-oop/python-methods/) e.g., log the exception, format the output, etc.
* Overriding the \_\_str\_\_ and \_\_repr\_\_ methods
* And doing anything else that you can do with regular classes.

In practice, you’ll want to keep the custom exceptions organized by creating a custom exception hierarchy. The custom exception hierarchy allows you to catch exceptions at multiple levels, like the standard exception classes.

In the MyCustomError class above, there are two magic methods, \_\_init\_\_ and \_\_str\_\_ that are automatically called during the exception handling process. Init is the method called when an instance is created and str is the magic method called when an instance is printed. Therefore, when an exception is raised, they are usually called in close succession. The raise statement in Python puts the programs into an error condition.

**À quoi sert \_\_str\_\_ ?**

La méthode spéciale **\_\_str\_\_** permet d'indiquer la représentation en chaîne de caractères d'un objet.

Cette chaîne de caractères est retournée lors de l'utilisation de la fonction **str** sur un objet ou lors de l'utilisation de la fonction **print** :

*import datetime*

*now = datetime.datetime.now()*

*print(now)*

*print(str(now))*

output :2022-07-03 11:31:49.179142

2022-07-03 11:31:49.179142

Pour implémenter cette méthode, il suffit de la surcharger à l'intérieur d'une classe :

*class Utilisateur:*

*def \_\_init\_\_(self, prenom, nom):*

*self.prenom = prenom*

*self.nom = nom*

*def \_\_str\_\_(self):*

*return self.prenom + " " + self.nom*

*utilisateur\_01 = Utilisateur(prenom="Patrick", nom="Smith")*

*print(utilisateur\_01)*

*print(str(utilisateur\_01))*

*output : Patrick Smith*

*Patrick Smith*

Cette méthode doit obligatoirement retourner une chaîne de caractères.

Si vous n'implémentez pas cette méthode dans votre classe, la méthode **\_\_repr\_\_** sera utilisée à la place.

*class Utilisateur:*

*def \_\_init\_\_(self, prenom, nom):*

*self.prenom = prenom*

*self.nom = nom*

*# On implémente la méthode \_\_repr\_\_ mais pas \_\_str\_\_*

*def \_\_repr\_\_(self):*

*return "Utilisateur(prenom='{}', nom='{}')".format(self.prenom, self.nom)*

*utilisateur\_01 = Utilisateur(prenom="Patrick", nom="Smith")*

*"""Quand on fait un print de l'objet ou qu'on utilise la fonction str,*

*le résultat de \_\_repr\_\_ nous est retourné"""*

*print(utilisateur\_01)*

*print(str(utilisateur\_01))*

*output : Utilisateur(prenom='Patrick', nom='Smith')*

*Utilisateur(prenom='Patrick', nom='Smith')*

Si vous n'avez pas implémenté la méthode **\_\_repr\_\_**, la représentation par défaut de la classe **object** (dont toutes les classes héritent) sera utilisée :

*# On n'implémente ni \_\_repr\_\_ ni \_\_str\_\_*

*class Utilisateur:*

*def \_\_init\_\_(self, prenom, nom):*

*self.prenom = prenom*

*self.nom = nom*

*utilisateur\_01 = Utilisateur(prenom="Patrick", nom="Smith")*

*# Quand on utilise str sur notre instance, la classe et l'adresse en mémoire de l'instance sont affichées*

*print(utilisateur\_01) # <\_\_main\_\_.Utilisateur object at 0x10f5952e0>*

*print(str(utilisateur\_01)) # <\_\_main\_\_.Utilisateur object at 0x10f5952e0>*

*# Cette implémentation provient de la classe object*

*print(object.\_\_repr\_\_(utilisateur\_01)) # <\_\_main\_\_.Utilisateur object at 0x10f5952e0>*

(si vous exécutez le code ci-dessus, l'adresse en mémoire n'est pas affichée car le code est exécuté dans votre navigateur internet)

Pour résumer tous les cas de figure possibles :

class Bonjour:

def \_\_repr\_\_(self):

return "repr de Bonjour"

def \_\_str\_\_(self):

return "str de Bonjour"

b = Bonjour()

print("Avec \_\_repr\_\_ et \_\_str\_\_ :")

print("print(b) -> ", b)

print("repr(b) -> ", repr(b))

print("str(b) -> ", str(b))

print("-"\*50)

class Bonjour:

def \_\_str\_\_(self):

return "str de Bonjour"

b = Bonjour()

print("Avec \_\_str\_\_ uniquement :")

print("print(b) -> ", b)

print("repr(b) -> ", repr(b))

print("str(b) -> ", str(b))

print("-"\*50)

class Bonjour:

def \_\_repr\_\_(self):

return "repr de Bonjour"

b = Bonjour()

print("Avec \_\_repr\_\_ uniquement :")

print("print(b) -> ", b)

print("repr(b) -> ", repr(b))

print("str(b) -> ", str(b))

print("-"\*50)

class Bonjour:

pass

b = Bonjour()

print("Avec aucune implémentation :")

print("print(b) -> ", b)

print("repr(b) -> ", repr(b))

print("str(b) -> ", str(b))

output :

Avec \_\_repr\_\_ et \_\_str\_\_ :

print(b) -> str de Bonjour

repr(b) -> repr de Bonjour

str(b) -> str de Bonjour

--------------------------------------------------

Avec \_\_str\_\_ uniquement :

print(b) -> str de Bonjour

repr(b) -> <\_\_main\_\_.Bonjour object at 0x000001D483B01C90>

str(b) -> str de Bonjour

--------------------------------------------------

Avec \_\_repr\_\_ uniquement :

print(b) -> repr de Bonjour

repr(b) -> repr de Bonjour

str(b) -> repr de Bonjour

--------------------------------------------------

Avec aucune implémentation :

print(b) -> <\_\_main\_\_.Bonjour object at 0x000001D483B01C90>

repr(b) -> <\_\_main\_\_.Bonjour object at 0x000001D483B01C90>

str(b) -> <\_\_main\_\_.Bonjour object at 0x000001D483B01C90>