Python avancé

Les Annotations

1. Introduction - Le typage en programmation

Lorsque l'on apprend à coder en Python, la question du **typage** est une question que l'on ne considère que très rarement alors que celle-ci est incontournable avec un langage comme Java. D'où vient donc cet "oubli" de la part d'un des langages de programmation les plus utilisés du moment ? Pour répondre à cette question, il est important de creuser quelques notions de base.

Le **typage en programmation** correspond simplement à la définition de la nature des valeurs que peut prendre les données que l'on manipule. Pour tester le type d'une variable en Python, on fait appel à la fonction `type()` qui retourne, sans surprise, le type de l'objet entré en paramètre de la fonction. La fonction s'utilise comme suit :

```
a = 2
type_de_a = type(a)
```

Le type de l'argument s'affiche à l'aide de la fonction `print` sous la forme :

```
>>> <class 'int'>
```

Dans l'exemple ci-dessus, on teste une variable numérique entière, le type correspondant est `int`. En Python, il existe une multitude de types dits natifs, vous trouverez la liste exhaustive sur le site de la documentation officielle (https://docs.python.org/fr/3/library/stdtypes.html).

- (a) Définissez une nouvelle variable appelée **b** dont le type sera `str`.

In [7]:

(/dashboard)

```
# Insérez votre code ici
b = str()

print(type(b))
```

RUN CELL RUN ALL ( ) (834) (831)

(/hub/dashboard)

```
<class 'str'>
```

UNVALIDATE SAVE Robin BIRON ()

Hide solution

(/hub/cheatsheet)

In [4]:

(/hub/progress)

```
b = 'Hello World'
print(type(b))
```

```
<class 'str'>
```

(/typo)

(/forum/271)

Il existe plusieurs types de typages en programmation. On en distingue principalement deux : le **typage statique** et le **typage dynamique**.

Typage statique

On appelle un langage à **typage statique**, un langage dans lequel chaque variable doit être assignée à un type précisé par le programmeur. Cette technique est adopté par de nombreux langages de programmation tels que Java, C ou C++.
Cette approche du code offre quelques avantages dont notamment une certaine rigueur dans l'écriture et la définition des variables : il n'y a pas de doute sur la nature des variables et il devient d'autant plus facile pour l'interpréteur de repérer des erreurs liées au type.

Prenons l'exemple de Java pour la définition d'une variable numérique, la syntaxe est la suivante :

```
int a = 2;
float b = 3.;
String c = 'Hello World';
```

Pas besoin d'avoir une maîtrise de Java pour comprendre cette syntaxe. Chaque variable **a**, **b** ou **c** est introduite par le **type** de celle-ci et est suivie par la **valeur attribuée**. L'aspect statique du typage apparaît nettement et pour le vérifier, il suffit simplement de voir ce qu'il se passerait si l'on essayait d'assigner une nouvelle valeur à une des variables. Vous pouvez tentez par vous même sur ce compiler (https://repl.it/languages/java10) en ligne.

Comme dit précédemment, l'un des avantages de ce typage est la **rigueur** qu'il entraîne, mais cela facilite également **la lecture du code** par une personne autre que le programmeur originel, ce qui est extrêmement intéressant dès lors que celui-ci écrit du code pour une entreprise. Par ailleurs, un autre point qu'il est important de mentionner est que la

vérification de type est généralement faite lors de la **compilation** du code. Cela veut dire que l'exécution de ce dernier peut se faire à **pleine vitesse** ce qui n'est pas le cas pour un langage typé dynamiquement. En revanche, cette approche rend la programmation nettement plus rigide voire même fastidieuse par moment, ce qui peut en pousser plus d'un à s'orienter vers un langage de programmation à **typage dynamique**.

Typage dynamique

Contrairement à Java, Python a opté pour un typage dit **dynamique**. Concrètement, cela veut dire qu'à la différence d'un langage statique, le typage n'est réalisé et vérifié qu'après l'exécution du code et pas avant. Outre une perte relative de vitesse d'exécution, qu'est-ce que cela implique ?

- Lorsque l'on définit une variable en Python, nous ne sommes pas obligés de préciser le type de celle-ci, il est reconnu par l'interpréteur.

Exemple :

```
variable = 'chaîne de caractère'
print(type(variable))

--- Exécution ---

>>> <class 'str'>
```

- Une fois une variable définie, il est tout à fait possible de lui assigner une nouvelle valeur, peu importe son type, sans causer d'erreur.

Exemple :

```
variable = 'chaîne de caractère'
print(type(variable))
variable = 3
print(type(variable))

--- Exécution ---

>>> <class 'str'>
>>> <class 'int'>
```

Ce mécanisme peut paraître un peu contre-intuitif lorsque l'on est habitué à la rigidité usuelle de l'informatique, mais lorsque l'on sait comment Python gère le stockage en mémoire des variables, cela devient vite assez clair. Pour s'en rendre compte, on utilisera la fonction `id` de Python qui renvoie l'identifiant de la localisation en mémoire de l'objet pris en argument.

- (b) Définissez deux variables toutes deux égales à une même valeur numérique.
- (c) À l'aide de la fonction `id`, affichez les localisations des deux variables crées, ainsi que celle de la valeur numérique choisie.

- (d) Commentez.

RUN CELL RUN ALL

In [13]:

```
# Insérez votre code ici

a=2
b=2

print(id(a),'\n',id(b))
```

UNVALIDATE SAVE

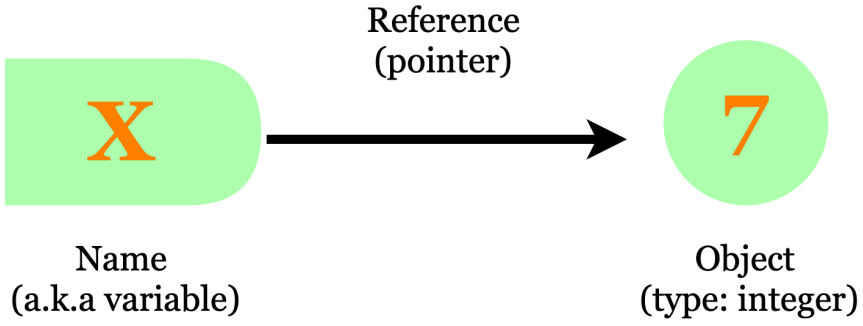

```
94809573329472
 94809573329472
```

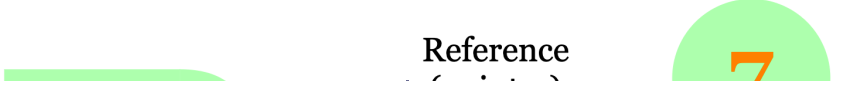
Hide solution

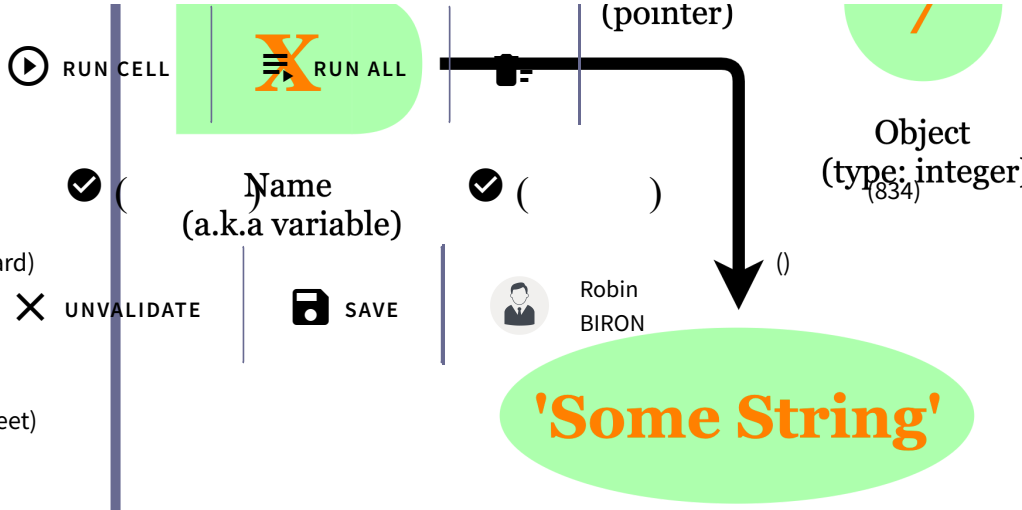
In [ ]:

```
variable_1 = 5

variable_2 = 5

print(id(5))

print(id(variable_1))

print(id(variable_2))
```

On constate que les identifiants sont les mêmes, autrement dit, toutes ces objets sont stockés au **même endroit**. Informatiquement, sont-ils donc parfaitement égaux ? Pas tout à fait. En réalité Python alloue de la place en mémoire à des objets tels que des valeurs numériques, des chaînes de caractères... et la variable n'est qu'un *raccourci* qui permet de pointer vers cet objet.







(834) (831)

On comprend donc déjà mieux comment le typage dynamique fonctionne en Python, pour réattribuer une valeur à une variable, il s'agit donc simplement de **rediriger** le pointeur de cette variable vers le nouvel objet que l'on décide de lui assigner.

Typage dynamique ou typage statique ?

À cette question, il n'y a pas de réponse fixe: c'est défini par le cadre de programmation. Si la mission exige une **rigueur** et une **transparence absolue** dans le code alors on optera pour un **typage statique**. Si le programmeur se voit laisser la main libre sur le code alors privilégier la **flexibilité** et la **rapidité d'écriture** d'un **typage dynamique** peut s'avérer être la solution intéressante.
Mais alors, si l'on veut un typage statique, faut-il nécessairement éviter Python ?

2. Les annotations ou le typage statique selon Python

Pour simuler un typage statique, Python propose un système d'**annotations**, qui permet à l'utilisateur de préciser le type des variables que l'on souhaite

qui permet à l'utilisateur de préciser le type des variables que l'on souhaite avoir en argument d'une fonction ainsi le type voulu en sortie d'une fonction.

Les annotations s'utilisent comme suit :

```
def une_fonction(a : str='Hello World') -> None :
    print(a)
```

La fonction que l'on définit en exemple est extrêmement simple. Cette fonction qui prend en argument une chaine de caractère `a`, valant par défaut `'Hello World'` et qui affiche cette chaîne de caractère en sortie.
À cette définition, on précise deux annotations.

- La première annotation correspond au type de la variable souhaitée en argument, `a : str`. Ici, on **indique** que l'argument `a` entré par l'utilisateur de notre fonction doit être de type `str`.
- La deuxième annotation indique le type de la valeur en sortie de notre fonction, `-> None`. Ici, on **indique** que la fonction *retourne* rien, ce qui est cohérent avec son objectif d'*afficher* uniquement un résultat.

Ces annotations sont accessibles à l'aide de l'attribut `__annotations__`, dans le cas de notre exemple, on aura les annotations suivantes :

```
print(une_fonction.__annotations__)

--- Exécution ---

>>> {'a': str, 'return': None}
```

- (a) Définissez une nouvelle fonction qui calculera l'aire d'un rectangle. Elle prendra en argument la **longueur** et la **largeur** du rectangle, et retournera l'aire de ce dernier. Les variables de la fonction ainsi que son résultat seront annotés du type `float`.
- (b) Affichez les annotations de la fonction.

```
# Insérez votre code ici
def aire_rectangle(longueur : float, largeur : float) -> float:
    return longueur*largeur

print(aire_rectangle.__annotations__)
```

```
{'longueur': <class 'float'>, 'largeur': <class 'float'>, 'r
urn': <class 'float'>}
```

Hide solution

In [ ]:

```
def aire( l : float, L : float ) -> float :
    return l*L

print(aire.__annotations__)
```

> Néanmoins, ces annotations ne sont que des annotations et non pas des déclarations de type. En effet, cela ne suffit pas à changer la nature fondamentalement dynamique du typage en Python. En réalité, on peut même se permettre d'annoter nos fonctions avec ce que l'on veut sans enfreindre la bonne exécution de celle-ci.

- (c) Définissez une fonction **afficher**, qui prendra en argument une chaîne de caractères et qui affichera, sans retourner, cette dernière.
- (d) Annotez la fonction.
- (e) Exécutez la fonction avec un argument de type autre que `str`. Commentez.

In [13]:

```
# Insérez votre code ici
def afficher(a : str) -> None:
    print(a)

print(afficher.__annotations__)

afficher(3)
```

```
{'a': <class 'str'>, 'return': None}
3
```

Hide solution

In [11]:

```
def afficher(variable : str) -> str:
    print(variable)

print(afficher.__annotations__)
afficher(3)
```

RUN CELL RUN ALL (834) (831)

{'variable': <class 'str'>, 'return': <class 'str'>}
3

UNVALIDATE SAVE Robin BIRON

Vous l'aurez remarqué, la fonction ne renvoie aucune erreur même si le type de l'argument entré ne correspond pas au type annoté au préalable. Encore une fois, ce système d'annotation n'est en réalité qu'un système **d'indications**, libre à l'utilisateur de suivre ces indications ou non. Toutefois, il existe un outil tiers qui permet d'effectuer la vérification de type et de permettre à Python de bénéficier d'une vérification de type comme un vrai langage statique.

3. MyPy

MyPy (https://mypy.readthedocs.io/en/stable/) est une librairie Python développée pour permettre à un utilisateur de vérifier le typage statique d'un code. Elle fonctionne de paire avec les annotations et s'assurent donc que celles-ci sont bien respectées. À défaut de rendre *invalide* le code si le typage n'est pas respecté, MyPy renvoie un **rapport détaillé** des erreurs rencontrés même si celui s'exécutera toujours si l'erreur n'est pas plus profonde.
La façon la plus courante de l'utiliser est de l'employer comme un débugger selon le schéma suivant :

1. Rédiger son code Python et l'enregistrer comme un fichier `.py`.
1. Sur un terminal, entrer la commande suivante : `mypy mon_fichier.py`

Si des erreurs sont détectées elles seront renvoyées par MyPy précisant le **type** de l'erreur, sa **position** dans le code ainsi que la **cause** de l'erreur.

Pour faire fonctionner MyPy dans un jupyter notebook, nous allons définir un `magic`. Nous ne rentrerons pas en détail là dessus, sachez juste que cela permet d'incorporer les fonctionnalités de vérification de type proposés par MyPy lors de l'exécution des cellules Jupyter.

- (a) Exécutez la cellule suivante pour instancier MyPy sur Jupyter.

In [21]:

```
from IPython.core.magic import register_cell_magic
@register_cell_magic
def typecheck(line, cell):
    from IPython import get_ipython
    from mypy import api
    cell = '\n' + cell
    mypy_result = api.run(['-c', cell] + line.split())
    if mypy_result[0]:
        print(mypy_result[0])
    if mypy_result[1]:
        print(mypy_result[1])
    shell = get_ipython()
    shell.run_cell(cell)
```

> Dès lors que cette cellule est lancée, il suffit de faire paraître `%%typecheck` au début de chacune des cellules dont on souhaite vérifier le type.

- (b) Définissez une fonction qui prendra en argument une liste et qui retournera une nouvelle liste à laquelle on ajoute un élement au choix. Précisez les annotations correspondantes.
- (c) Exécutez la fonction avec en argument une liste quelconque et en faisant attention à bien mentionner le magic de vérification de type.

In [22]:

```
%%typecheck
def my_function(l:list, element) -> list:
    l.append(element)
    return l


my_function(l=[5,2,6,"hello"], element="ok")
```

```
Success: no issues found in 1 source file
```

Out[22]:

```
[5, 2, 6, 'hello', 'ok']
```

Hide solution

In [23]:

```
%%typecheck
RUN CELL  RUN ALL

def function(L : list) -> list:
    L = L + ['2']
    return(L)

print(function(['1+1 ='']))
UNVALIDATE  SAVE
```

( ) ( ) (834) (831)

() Robin BIRON

```
Success: no issues found in 1 source file

['1+1 =', '2']
```

> On obtient, en plus du retour de notre fonction, un message qui atteste du bon fonctionnement de notre typage.
> Essayons de voir le cas de figure contraire, celui où le typage n'aurait pas été valide.

- (d) Définissez une nouvelle fonction `double` qui prendra en argument un entier (type `int`) et retournera le double de cet entier (type `int`).
- (e) À l'aide de cette fonction, afficher le double de 27.6. Observez.

In [29]:

```
%%typecheck

def double(a:int) -> int:
    return a*2

double(27.6)
```

```
<string>:6: error: Argument 1 to "double" has incompatible t
e "float"; expected "int"
Found 1 error in 1 file (checked 1 source file)
```

Out[29]:

```
55.2
```

Hide solution

In [25]:

```
%%typecheck
RUN CELL   RUN ALL
def double(a : int) -> int:
    a = a*2
    return(a)

double(27.6)
```

( ) ( ) (834) (831)

()

UNVALIDATE SAVE Robin BIRON

```
<string>:7: error: Argument 1 to "double" has incompatible t
e "float"; expected "int"
Found 1 error in 1 file (checked 1 source file)
```

Out[25]:

55.2

> Bien que la fonction s'exécute et retourne la valeur attendue, MyPy nous renvoie un message d'erreur indiquant que le typage annoté n'a pas été respecté. Ce message se lit de la façon suivante :
>
> - **La position de l'erreur** désignée par le numéro de la ligne où elle est repérée, ici elle correspond au numéro de la ligne où l'on exécute notre fonction `double()`.
> - **La cause de l'erreur**, ici il s'agit de l'incompatibilité entre le type du premier argument entré, `float`, et le type de l'argument attendu, `int`.
> - **Un récapitulatif** du nombre d'erreurs recensées par MyPy.
>
> Ce type d'erreur rapporté par la librairie est le plus classique, mais il en existe d'autres auxquels on peut s'attendre.

- (f) Toujours à partir de la fonction `double()`, exécutez celle-ci en prenant comme argument un vecteur numérique quelconque appartenant à la classe numpy.array. N'oubliez pas d'importer le package correspondant.

In [31]:

```
%%typecheck
import numpy as np

def double(a: int) -> int:
    a = a*2
    return(a)
a = np.array([1,5,4])
double(a)
```



```
<string>:3: error: Skipping analyzing 'numpy': found module
t no type hints or library stubs
<string>:3: note: See https://mypy.readthedocs.io/en/latest/ru
nning_mypy.html#missing-imports (https://mypy.readthedocs.io/e
n/latest/running_mypy.html#missing-imports)
Found 1 error in 1 file (checked 1 source file)
```

Out[31]:

```
array([ 2, 10,  8])
```

Hide solution

In [32]:

```
%%typecheck

import numpy as np

vec = np.array([2,4,6])

double(vec)
```

```
<string>:3: error: Skipping analyzing 'numpy': found module
t no type hints or library stubs
<string>:3: note: See https://mypy.readthedocs.io/en/latest/ru
nning_mypy.html#missing-imports (https://mypy.readthedocs.io/e
n/latest/running_mypy.html#missing-imports)
<string>:7: error: Name 'double' is not defined
Found 2 errors in 1 file (checked 1 source file)
```

Out[32]:

```
array([ 4,  8, 12])
```

Ici, on remarque que deux "erreurs" sont retournées alors qu'on ne s'attendait qu'à une sur l'incompatibilité de type.

- La première correspond au fait que les annotations de type ('type hints') **ne sont pas fournies** avec la version de numpy que nous avons importé.

  Par conséquent, bien que MyPy reconnaisse l'import de numpy dans notre cellule, il n'est pas en mesure de **reconnaitre les nouveaux types**

qui accompagnent cette librairie, dont les types `numpy.array`. Pour palier à ce problème soit il est possible de télécharger indépendamment les annotations des nouveaux types qu'introduisent la librairie soit ces annotations sont disponibles dans des versions plus récentes de la librairie en question.

- La deuxième erreur vient du fonctionnement même du magic MyPy, cette erreur n'aurait pas été levée lors d'une utilisation "classique" de la librairie, donc sans passer par un magic Jupyter. En effet, bien que l'on ait défini sans problème la fonction `double()` dans une cellule plus haut, celle-ci n'est plus reconnue par MyPy dans une nouvelle cellule. Une utilisation optimale de MyPy sur Jupyter serait donc de **regrouper tout notre code dans une seule et même cellule** afin de vérifier le typage efficacement sans biaiser le rapport fourni par de fausses erreurs telles que celle-ci.

Conclusion

- En programmation, on distingue deux méthodes de typages, le **typage statique** rigide et rigoureux, et le **typage dynamique**, simple et flexible.
- En Python natif, le typage est dynamique mais l'on peut tout de même se rapprocher d'un typage statique à l'aide des **annotations** qui fournissent une aide à la lecture et au développement en statique légèrement plus rigoureuse. Néanmoins les erreurs d'annotations restent indétectables.
- Pour palier à ceci, il existe des outils de détection de type tels que **MyPy** qui, couplé au système natif d'annotations, permet de détecter les erreurs de typage et ainsi rendre le code en Python plus propre. Cette méthode reste limitée contrairement à un typage statique natif.
- De nombreuses libraries utilisent les annotations pour permettre, par exemple, de générer une documentation de manière automatique.

✖ Unvalidate