Langages de script (Python)
CMTP n° 8 : Typage

I) Typage

Python est un langage de programmation à typage dynamique : les erreurs de type sont détectées par l'interpréteur pendant l'exécution et signalée avec des exceptions de la classe `TypeError`. Cependant, à partir des versions récentes du langage Python support les *annotations de type* et dispose d'*analyseurs statiques* pour la bonne utilisation des types qui reposent sur un système de type basé sur la notion de *gradual typing* (voir : https://en.wikipedia.org/wiki/Gradual_typing). Dans ce sujet nous allons explorer le typage en Python, via les annotations de type et l'analyseur statique `mypy`.
Pour nos premiers pas avec le typage en Python nous allons suivre ensemble le tutoriel officiel de `mypy` : https://mypy.readthedocs.io/en/stable/getting_started.html.

Exercice 1 : Setup
Installez `mypy` dans un virtualenv Python et lancez-le sur un ou plusieurs programmes Python que vous avez écris dans le passé. Essayez ensuite de lancer l'exécutable `mypy` sur un ou plusieurs programmes Python que vous avez écris dans le passé.

Une fonction Python qui ne contient pas des annotations de type reste *typée dynamiquement* et sera (sauf erreurs de type flagrantes) ignoré par `mypy`, p.ex. :

```
$ cat bad.py
def greeting(name): # dynamically typed (no type annotations)
    return 'Hello ' + name

greeting("Alice") # OK
greeting(42) # type error not detected by mypy

$ mypy bad.py
Success: no issues found in 1 source file

$ python3 bad.py
...
TypeError: can only concatenate str (not "int") to str
```

Vous pouvez annoter la fonction greeting pour indiquer qu'elle attend un argument de type string (`str`) et qu'elle retourne une valeur du même type. Cela rend la fonction *typée statiquement* et permet à `mypy` des reconnaître certains usages incorrectes de la fonction :

```
$ cat bad2.py
def greeting(name: str) -> str: # statically typed
    return 'Hello ' + name

greeting("Alice") # OK
greeting(42) # type error detected by mypy

$ mypy bad2.py
bad2.py:5: error: Argument 1 to "greeting" has incompatible type "int"; expected "str"
Found 1 error in 1 file (checked 1 source file)
```

Le principe derrière le *gradual typing* est de pouvoir mélanger du code typé dynamiquement avec du code typé statiquement. Plus vous ajoutez des annotations des types à vos fonctions, plus `mypy` sera capable de détecter statiquement des erreurs de typage.
Voici d'autres exemples d'annotations de type, notamment pour les fonctions qui peuvent retourner `None`, pour les arguments avec des valeurs par défaut

```
def p() -> None:
    print('hello')

a = p() # Error: "p" does not return a value
# ---------------------------------------------------------------------

def f():
    1 + 'x' # No type error, because f is dynamically typed

def g() -> None:
    1 + 'x' # Type error, because f is statically typed (due to annotation)
# ---------------------------------------------------------------------

def greeting(name: str, excited: bool = False) -> str:
    # "excited" is an optional argument; when given it must be a boolean
    message = 'Hello, {}'.format(name)
    if excited:
        message += '!!!'
    return message
```

Si vous ne connaissez pas le nom exact d'un type, vous pouvez toujours utiliser la fonction `type()` pour le découvrir interactivement.

```
type(A)
str
type(3)
int
type(3.14)
float
# etc.
```

Les types conteneurs (listes, tuples, ensemble, ...) peuvent aussi être annotée avec des *types génériques*, en utilisant les crochets pour indiquer le type (uniforme !) de leur contenu :

```
def greet_all(names: list[str]) -> None:
    for name in names:
        print('Hello ' + name)

names = ["Alice", "Bob", "Charlie"]
ages = [10, 20, 30]

greet_all(names) # Ok!
greet_all(ages)  # Error due to incompatible types
```

Attention : la syntaxe `list[str]`, avec "l" minuscule, est supportée à partir de la version 3.9 de Ptyhon. Avec les versions antérieures la syntaxe à utiliser pour les types conteneurs standards est `List[str]`, avec "L" majuscule, précédé par `from typing import List` au début du fichier.
Le module `typing` contient aussi d'autres types complexes, comme p.ex. le *type union* et le type optionnel (pour quelque chose qui peut être soit `None` soit une valeur d'un type donné) :

```
from typing import Optional

def greeting(name: Optional[str] = None) -> str:
    # Optional[str] means the same thing as Union[str, None]
    if name is None:
        name = 'stranger'
    return 'Hello, ' + name
```

```python
from typing import Union

def normalize_id(user_id: Union[int, str]) -> str:
    if isinstance(user_id, int):
        return 'user-{}'.format(100000 + user_id)
    else:
        return user_id


# same, but using type aliases to factorize frequently used types:

UserId = Union[int, str]

def normalize_id(user_id: UserId) -> str:
    if isinstance(user_id, int):
        return 'user-{}'.format(100000 + user_id)
    else:
        return user_id
```

Au fur et à mesure que vous ajoutez des annotations de types, `mypy` pourrait vous obliger à ajouter des annotations de types sur certaines variables, p.ex. :

```python
my_global_dict = {} # Error: Need type annotation for 'my_global_dict'

# If you're using Python 3.9+, lower case "d"
my_global_dict: dict[int, float] = {} # dictionary with int keys and float values

# If you're using Python 3.6+, upper case "D"
from typing import Dict
my_global_dict: Dict[int, float] = {}
```

Pour ce qui concerne la *programmation à objet*, chaque nom de classe peut être utilisé pour typer ces instances, et la hiérarchie des classes est bien respectée dans l'analyse de type.
Des types génériques sont aussi disponibles pour des abstractions typiques du langage, comme les iterables :

```python
from collections.abc import Iterable # or "from typing import Iterable"

def greet_all(names: Iterable[str]) -> None:
    for name in names:
        print('Hello ' + name)
```

Notez la différence entre le *type abstrait* "iterable" et les *types concrets* qui les réalisent, comme `list`, `set`, etc. Vous pouvez toujours typer une liste comme un iterable, mais pas le contraire. La règle d'or pour les annotations de type dans ce cas est de typer les valeurs de retours de fonctions avec le type plus spécifique possible et les arguments avec le type le plus générique possible. Donc si votre fonction ne nécessite pas d'une vraie liste pour marcher, mais fonctionnerait sur n'importe quel iterable, il faut bien la typer avec `Iterable`, comme dans l'exemple ci-dessus.
Pour plus d'informations, et pour tous les cas compliqués que vous allez certainement rencontrer pendant l'ajoute des informations de types à vos programmes, consultez :

- le "cheat sheets" de `mypy` : https://mypy.readthedocs.io/en/stable/index.html#overview-cheat-sheets
- et la guide de référence du système de type : https://mypy.readthedocs.io/en/stable/index.html#overview-type-system-reference

Exercice 2 : Typage de code existante

Reprenez vos solutions pour les exercices suivants des séances passées : [1]

1. si vous ne les aviez pas implementées à l'époque, implémentez les maintenant !

- CMTP 2 : Exercice 5 (“mots sans cube”)
- CMTP 2 : Exercice 6 (“bon parenthèse”)
- CMTP 3 : Exercice 1 (“ensembles”), toutes les fonctions
- CMTP 3 : Exercice 2 (“dictionnaires”), toutes les fonctions
- CMTP 5 : Exercice 2 (“Les prénoms en France 1”), toutes les fonctions
- CMTP 6 : (“mini-projet”), tous vos modules
- CMTP 7 : Exercice 5 (“itinéraire du métro”)

Pour chaque solution, sauvegardez la dans un fichier `.py` dédié si cela n'est pas déjà le cas, et validez le fichier avec `mypy` ; dans la plus part des cas aucun erreur devrait être détecté, mais seulement car la totalité de votre code reste pour le moment typé dynamiquement.
Ajoutez donc les annotations de type à toutes vos fonctions, pour le rendre typées statiquement. Relancez ensuite `mypy` sur votre code (vous pouvez utiliser l'option `--disallow-untyped-defs` de `mypy` pour vous assurez que vous annoté toutes les definitions de votre fichier) et vérifiez qu'il n'y a toujours pas des erreurs de typage ou, les cas échéant, corrigez les.

II) Règles de codage

Le typage est juste une parmi plusieurs formes d'analyse statique qui sont couramment utilisées dans le développement en Python. Notamment, une *norme de codage* existe en Python, qui concerne comment syntactiquement écrire du code Python “standard”. Elle se trouve dans le document *PEP 8 — Style Guide for Python Code* (souvent appelée seulement “PEP8”), disponible ici : https://www.python.org/dev/peps/pep-0008/. Cette norme concerne l'indentation, le nommage des identifiants, les commentaires, etc.
Comme toutes autres règles de codage, il peut devenir assez pénible de devoir les respecter à la main, alors que les avantages de le faire bénéficient tous les développeurs qui collaborent au même projet. Des outils automatiques existent donc pour vérifier statiquement que votre code respect cette norme, comme notamment `Flake8` (à prononcer comme “flaky”) : https://flake8.pycqa.org/.

Exercice 3 : Flake8

Installez l'outil `flake8` dans votre virtualenv.
Reprenez ensuite le code des exercices déjà évoqués dans l'exercice 2. Lancez flake8 sur ce code, combien d'erreurs sont identifiés ? Corrigez les tous.
Si vraiment vous ne voulez pas corriger certains erreurs, faites en sorte que `flake8` ignore le problème *le plus spécifique possible* avec la syntaxe dédiée dans le commentaires, p.ex. :

```
example = lambda: 'example' # noqa: E731
```

Pour plus d'information sur ce point : https://flake8.pycqa.org/en/latest/user/violations.html#ignoring-violations-with-flake8
Voyez vous des erreurs récurrents de non respect de la norme PEP8 dans votre code, que vous pouvez éviter dans le futur ?

Une alternative au respect manuel des règles de codage est le formatage automatique de code. Un outil populaire qui répondre à ce besoin dans le cas de Python est Black : https://black.readthedocs.io/. Black peut être utilisé à la fois pour *vérifier* que votre code respecte ses règles de codage (qui sont compatibles, mais plus précises que PEP 8) et pour *formater automatiquement* vos fichiers Python et les rendre conformes (entre autres) à PEP 8.

Exercice 4 : Black

Installez l'outil `black` dans votre virtualenv.
Reprenez à nouveau le code des exercices 2 et 3 précédents. Utilisez `black --check` sur ces fichiers pour vérifier s'ils respectez ou pas les règles de style de Black. Si cela n'est pas le cas (ce qui est fort probable), utilisez d'abord `black --diff` sur certains fichiers pour voir les différences de style que Black apporterez. Si vous êtes satisfait du résultat, formatez automatiquement tout vos fichiers avec black.

Beaucoup d'éditeurs pour Python supportent le formatage automatique (avec Black ou grâce à du support natif). Vérifier quel type de support votre éditeur préféré a pour cette fonctionnalité, et configurez le pour formater votre code Python automatiquement dans le futur, comme pour vous indiquer d'autres violation de PEP 8 pendant l'édition de code.