Le typage statique en Python : un retour d'expérience

Christian Poli, SED

Jean-Daniel Fekete, EPI Aviz

Demandez le programme, session du 27 juin 2023

Le contexte du projet **ProgressiVis**

ProgressiVis est une toolkit qui implémente un nouveau paradigme de programmation

- l'**analyse progressive de données**,
- pour visualiser et explorer interactivement des données de grande taille

Il permet de spécifier des traitements de données (dataflow) qui sont exécutés **progressivement**.

- Un résultat approximatif est visualisé initialement
- il s'améliore progressivement jusqu'au résultat final
- ou jusqu'à ce que l'utilisateur décide qu'il peut prendre une décision
- souvent avant la fin du calcul
- L'analyse progressive permet à l'analyste de contrôler le compromis temps/qualité

Pour implémenter ce paradigme, il faut (hélas) réimplémenter des bibliothèques standard Python afin que l'exécution soit progressive

C'est un gros travail et on a introduit le typage pour nous aider à gérer les évolutions de la toolkit.

Quelques généralités sur le typage

Les normes relatives au typage

- PEP 3107 Function Annotations
- PEP 483, PEP 484 Type hints
- PEP 526 Syntax for Variable Annotations
- PEP 561 Distributing and Packaging Type Information
- PEP 544 Protocols: Structural subtyping (static duck typing)
- PEP 585 Type Hinting Generics In Standard Collections
- PEP 647 User-Defined Type Guards

Programmes de vérification du typage (type checker)

- mypy, aujourd'hui le standard. On n'utilisera que lui !
- pytype, par Google. Fait de l'inférence de type.
- pyright, par Microsoft. Fait aussi de l'inférence de type.
- pyre, par Facebook. Plus proche de mypy.

Quelques généralités ... (suite)

Syntaxe

Exemple :

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

Alias de type simple :

```
Url = str

def retry(url: Url, retry_count: int) -> None: ...
```

... ou plus complexe :

```
from typing import TypeVar, Iterable, Tuple

T = TypeVar('T', int, float, complex)
Vector = Iterable[Tuple[T, T]]
```

[NOTE]: Les exemples précédents proviennent de la PEP-484

Quelques généralités ... (suite)

La généricité

- Suppose l'existence de paramètres, définis via la construction **TypeVar**.
- Indispensable à la définition des conteneurs typés
- Riche ensemble de conteneurs typés dans le module standard **typing**

```
from typing import Sequence, TypeVar

MonType = TypeVar('MonType')      # Declare type variable

def first(l: Sequence[MonType]) -> MonType:   # Generic function
    return l[0]
```

On peut définir des types génériques sur mesure en héritant de la classe `typing.Generic` mais dans *ProgressiVis* on n'a pas été confronté à ce besoin pour le moment.

On peut ne typer que certaines fonctions. Par défaut, une variable non typée a le type `Any`.

On peut aussi aider le type checker en utilisant des assertions, des casts et encore de TypeGuards.

[NOTE]: Les exemples précédents proviennent de la PEP-484

Quelques généralités … (suite)

Sur le (sous)typage

Deux sous-typages sont possibles :

- **nominal** : basé sur la hierarchie des classes
- **structurel** : équivalent statique du **duck typing**
  - un objet est du bon type, s'il possède les méthodes avec les bonnes signatures pour remplir un rôle
  - rappelle les *interfaces* dans *Java*
  - mise en oeuvre via des **protocoles** (cf. PEP-544)

Quelques généralités ... (suite)

Les protocoles

- Spécification via la PEP-544
- Les protocoles prédéfinis dans le module **typing** couvrent la quasi-totalité des besoins que nous avons rencontré (typiquement pour les conteneurs : Sequence, List, Mapping etc.)
- Définir ses propres protocoles est possible en dérivant la classe **typing.Prococol**

Quelques généralités ... (suite)

Ou trouver les annotations des types des dépendances ?

Actuellement les annotations sont fournies de 4 manières (cf. PEP-561):

- dans le code (solution préférée)
- **type stubs** inclus dans le package (ex: numpy)
- **type stubs** fournis par un package associé (ex: pandas-stubs pour pandas)
- **type stubs** dans le dépôt typeshed

NB: les packages implémentant le typage doivent l'indiquer via un marqueur (fichier vide) **py.typed** à la racine

Quand les annotations ne sont pas fournies on peut :

- Ignorer le problème (via le hint `# type: ignore`) si l'impact est limité
- Ecrire soi même les stubs manquants (juste le sous-ensemble indispensable) ce qui n'est pas difficile. Dans **ProgressiVis** nous avons eu à le faire pour plusieurs packages dépourvus de stubs jusqu'au présent : *pyarrow*, *dataketches*, *ipywidgets* etc.

Quelques généralités … (suite)

Les fichiers stub ont l'extension **.pyi**

Voici un fragment du stub écrit pour *pyarrow* dans **ProgressiVis** :

```
class Array:
    null_count: int
    type: DataType
    def is_null(self, x: Any = None) -> BooleanArray: ...
    def cast(self, typ: Any) -> Array: ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[Scalar]: ...
```

Aider mypy : assert

Le type checking de cette classe :

```
class BankAccount:
   def __init__(self) -> None:
       self.balance: Optional[float] = None

   def initialize(self, amount: float = 0) -> None:
       self.balance = amount

   def deposit(self, amount: float) -> None:
      self.balance += amount

   def first_deposit(self, amount: float)-> None:
       self.initialize()
       self.deposit(amount)
```

produit l'erreur suivante au niveau de la méthode `deposit()` :

```
(dlp2023) poli@pcpoli:~/DLP/dlp-python-typing-2023$ mypy examples/bank_account.py
examples/bank_account.py:14: error: Unsupported operand types for + ("None" and "float")  [operator]
examples/bank_account.py:14: note: Left operand is of type "Optional[float]"
Found 1 error in 1 file (checked 1 source file)
```

Alors que le code est sémantiquement valide. On peut aider mypy ainsi :

```
   def deposit(self, amount: float) -> None:
      assert self.balance is not None
      self.balance += amount
```

Aider mypy : TypeGuard

Par ailleurs, on peut trouver certaines vérifications récurentes trop verbeuses. Par exemple, au lieu d'écrire :

```
assert x is not None
```

ecrire plutôt :

```
def nn(x: Any)-> bool:
   return x is not None

assert nn(x)
```

Ce code ne fonctionne pas en l'état mais la **PEP 647** permet de le rendre valide via les **TypeGuard** :

```
def nn(x: Any)-> TypeGuard[bool]:
   return x is not None
```

et la fonction `deposit()` devient :

```
   def deposit(self, amount: float) -> None:
      assert nn(self.balance)
      self.balance += amount
```

Aider mypy : assert pour typer une `@property`

Une astuce pour limiter le nombre de **assert**

Une représentation simplifiée des modules ProgressiVis :

```
class PTable:
    """
    A ProgressiVis specific data structure similar to Python Pandas or R DataFrame, but
    column-based and supporting fast addition of items.
    """
    def __init__(self, dshape: str, *args: Any, **kw: Any) -> None:
        # ...
        self.dshape: str = dshape # ex: {a: int32, b: float64}

class PDict(dict[str, Any]):
    """
    A ProgressiVis specific dictionary, keeping the history of changes
    """
    ...

class Module:
    """
    ProgressiVis modules root
    """
    def __init__(self, *args: Any, **kw: Any) -> None:
        # ...
        self.result: PTable|PDict|None = None
```

Aider mypy : assert pour typer une `@property`

Ce module particulier utilise une table en sortie :

```python
class MyModule(Module):
    def __init__(self, dshape: str, *args: Any, **kw: Any) -> None:
        # ...
        self.result = PTable("{a: int32, b: float64}")

    def run(self) -> None:
        # ...
        print(self.result.dshape)
```

et sans surprise, mypy proteste :

```
(dlp2023) poli@pcpoli:~/DLP/dlp-python-typing-2023$ mypy --strict examples/module.py
examples/module.py:30: error: Item "PDict" of "Union[PTable, PDict, None]" has no attribute "dshape"  [union-attr]
examples/module.py:30: error: Item "None" of "Union[PTable, PDict, None]" has no attribute "dshape"  [union-attr]
Found 2 errors in 1 file (checked 1 source file)
```

Pour éviter des **assert** superflues dans tous les modules, on peut utiliser l'héritage et des **properties**

Aider mypy : assert pour typer une `@property`

Ici, la property **ptable** sert l'alias pour l'attribut **result** avec un typage retreci :

```
class Module:
    """
    ProgressiVis modules root
    """
    def __init__(self, *args: Any, **kw: Any) -> None:
        # ...
        self.result: PTable|PDict|None = None

    @property
    def ptable(self) -> PTable:
        assert isinstance(self.result, PTable)
        return self.result

class MyModule(Module):
    def __init__(self, dshape: str, *args: Any, **kw: Any) -> None:
        # ...
        self.result = PTable("{a: int32, b: float64}")

    def run(self) -> None:
        # ...
        print(self.ptable.dshape)
```

NB: Cette solution évite la prolifération des **assert** mais si les noms des alias sont mal choisis ça peut poser un problème de lisibilité

Aider mypy : cast

Le cast

Les classes **Teacher** et **Student** sont des sous-classes de **Member** :

```
class Member:
    def __init__(self, name: str) -> None:
        self.name = name

class Teacher(Member):
    def __init__(self, name: str, salary: float) -> None:
        super().__init__(name)
        self.salary = salary

class Student(Member):
    def __init__(self, name: str, marks: list[int]) -> None:
        super().__init__(name)
        self.marks = marks
```

Aider mypy : cast

Si une CLI utilise ces classes ainsi :

```
if __name__ == "__main__":
    if len(sys.argv) == 1:
        print("Use: {teacher|student} <name> {salary|marks}")
        sys.exit()
    assert len(sys.argv) > 3
    role: str = sys.argv[1]
    assert role in ("teacher", "student")
    name: str = sys.argv[2]
    member: Member
    if role == "teacher":
        member = Teacher(name, float(sys.argv[3]))
    else:
        member = Student(name, [int(m) for m in sys.argv[3:]])
    if isinstance(member, Student):
       if not len(member.marks):
           print("Marks list cannot be empty")
    if role == "teacher":
       if member.salary <= 0:
           print("Salary must be positive")
```

mypy est en difficulté :

```
(dlp2023) poli@pcpoli:~/DLP/dlp-python-typing-2023$ mypy examples/school.py
examples/school.py:34: error: "Member" has no attribute "salary"  [attr-defined]
Found 1 error in 1 file (checked 1 source file)
```

Aider mypy : cast

C'est dans ce contexte que le **cast** intervient :

```
from typing import cast
...
...
    if role == "teacher":
       if cast(Teacher, member).salary  <= 0:
           print("Salary must be positive")
```

NB: D'autres situations plus complexes, liées principalement aux **python magics**, peuvent se présenter. On verra un exemple plus loin.

Typage dans **ProgressiVis** : les motivations

- Fiabilisation du code existant:
  - Le typage peut mettre en évidence des incohérences ou des bugs passés inaperçues.
- Re-lecture plus aisée des parties non documentées du code (même de son propre code)
- Une aide essentielle à la documentation des APIs:
  - Sphinx/autodoc sont désormais en mesure d'exploiter les annotations des types
- Aide en ligne contextuelle (autocompletion ...) plus efficace avec certains éditeurs

Typage dans **ProgressiVis** : le processus

Le système de typage offre un évantail de possibilités pour une approche graduée.

Dans un premier temps nous avons mis en place un contrôle avec *mypy* en mode *non-strict*. Dans ce mode :

- les fonctions/méthodes pas encore typées sont ignorées par mypy
- les conteneurs peuvent être typés de manière plus "laxiste" (Ex: List au lieu de List[int])

Dans un deuxième temps, nous avons activé le mode *strict* :

- NB: si le passage au mode *strict* est trop dur on peut desactiver temporairement certaines options

Une fois le checking strict opérationnel, nous l'avons intégré à l'intégration continue à 2 niveaux :

- coté client : hook git via le module pre-commit
- coté serveur via un test lancé par github/Actions

Typage dans **ProgressiVis** : quelques expériences intéressantes

Dans la plus grande partie du code, le typage statique a pu être fait sans surprise particulière.

Pourtant, certains aspects du projet nous ont mis devant de petits défis techniques ou simples interrrogations qu'on a jugé dignes d'être mentionnés ici.

Ces défis ont été occasionnés par :

- les particularités de certaines dépendances de **ProgressiVis**, très importantes dans son fonctionnement:
  - numpy, pandas, pyarrow manipulant des tableaux et fichiers de données en différents formats
  - datasketches, scipy, scikit-learn, numexpr pour le calcul
  - ipywidgets, traitlets pour les interfaces graphiques **jupyter**
- l'emploi significatif de décorateurs dans la définition des artefacts du système

Le cas de numpy

le typage des données n'est pas statique :

```
>>> import numpy as np
>>> a = np.array([1, 2])
>>> a.dtype
dtype('int64')
>>> a.dtype = "float32"
>>> a
array([1.e-45, 0.e+00, 3.e-45, 0.e+00], dtype=float32)
>>> a.dtype = "bool"
>>> a
array([ True, False, False, False, False, False, False, False,  True,
       False, False, False, False, False, False, False])
>>>
```

Les (data)types numériques numpy ne sont pas forcément des types python :

```
>>> a = np.array([1, 2])
>>> a.dtype
dtype('int64')
>>> isinstance(a.dtype, int)
False
>>>
```

Le cas de numpy (suite)

Le règles de promotion des dtypes sont complexes. Pour palier ces problèmes, numpy propose désormais 2 alias:

- *ArrayLike*
- *DTypeLike*

Dans **ProgressiVis** nous avons préféré, au moins pour l'instant, utiliser des *np.ndarray[Any, Any]* ce qui revient à être :

- strict sur le typage des objets
- non strict sur le type des données

Le défi des décorateurs

Le décorateur en soi ne pose pas de problème particulier mais son action peut en poser

Quand un décorateur (en principe de classe) crée des méthodes, attributs etc. :

- les créations se produisent au chargement du module
- mypy (ou autre type checker), qui agit en amont, n'est pas au courant de ces créations

Le défi des décorateurs (suite)

Le phénomène existe déjà dans la bibliothèque standard:

```
@dataclass
class InventoryItem:
    '''Class for keeping track of an item in inventory.'''
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

Le décorateur `@dataclass` va créer (entre autres) le constructeur :

```
def __init__(self, name: str, unit_price: float, quantity_on_hand: int = 0) -> None:
    self.name = name
    self.unit_price = unit_price
    self.quantity_on_hand = quantity_on_hand
```

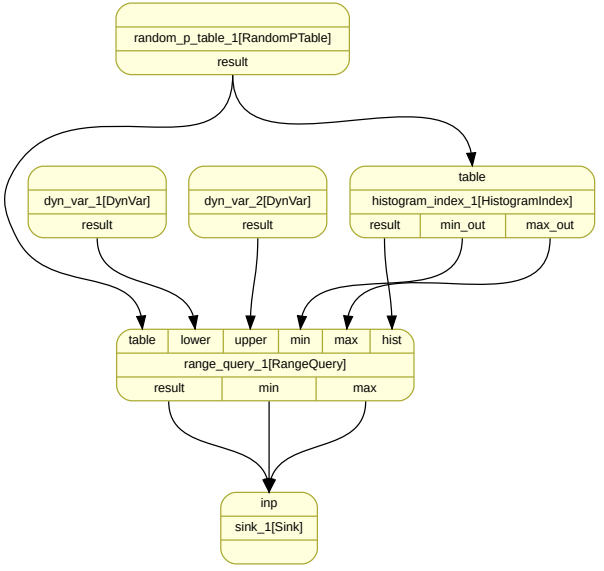
Ce constructeur n'existe pas pour mypy, qui fait une analyse syntaxique du code sans charger le module et exécuter le décorateur *@dataclass*

Pour gérer cette situation, mypy dispose d'un plugin dédié à `@dataclass`.

[NOTE]: Les exemples précédents proviennent de la PEP-557

Les décorateurs dans ProgressiVis

Un dataflow **ProgressiVis** est composé de modules interconnectés par slots:



Les décorateurs dans **ProgressiVis** (suite)

La "connectique" des modules est mise en oeuvre via des décorateurs:

```
@def_input("table", PTable)
@def_output("result", PTableSelectedView)
@def_output("min_out", PDict, required=False)
@def_output("max_out", PDict, required=False)
class HistogramIndex(Module):
     # Body
```

En particulier, le décorateur `@def_output` crée au chargement un attribut, ou plus précisement une `@property` avec setter que *mypy* ne peut pas connaître.

Pour l'aider, nous avons écrit un plugin.

Le plugin

```
from mypy.plugin import Plugin, ClassDefContext
from sqlmypy import add_var_to_class, CB  # type: ignore
from mypy.nodes import TypeInfo, NameExpr, StrExpr, CallExpr
from mypy.types import Instance, UnionType, NoneTyp
from typing import Any, Union, Optional, List, cast


def decl_deco_hook(ctx: ClassDefContext) -> None:
    # Body
    # ...
    # ...
    add_var_to_class(attr, typ, ctx.cls.info)
```

qui est activé via un hook :

```
class ModulePlugin(Plugin):
    def get_class_decorator_hook(self, fullname: str) -> CB[ClassDefContext]:
        if fullname == "progressivis.core.module.def_output":
            return decl_deco_hook
```

Ce qui ne facilite pas l'écriture des plugins est la documentation un peu trop succinte de l'API plugin de `mypy`.

Les ipywidgets et le typage statique

Les notebook Python peuvent utiliser des widgets standard, les **ipywidgets**, pour améliorer la présentation ou implémenter des interactions.

A l'heure actuelle :

- Le code python des **ipywidgets** n'est pas typé statiquement
- il n'existe pas de stubs officiels pour ipywidgets
- les widgets tiers basés sur la techologie **ipywidgets** qu'on a l'habitude d'utiliser ne sont pas typés non plus à l'heure actuelle

Notre constat : L'API actuel des widgets conteneurs (VBox, HBox, Tab, Accordion) rend le typage difficile dans le cas le plus général

Mais notre expérience nous a montré qu'on peut trouver des solutions de typage propres qui conviennent dans les situations les plus courantes

Concrètement, les widgets conteneurs héritent de la classe **Box** l'attribut **children** contenant les widgets qui les composent. Le stub suivant reflète cela:

```
class Box(DOMWidget):
    children: Sequence[DOMWidget]
    def __init__(self, *args: Any, **kw: Any) -> None: ...
```

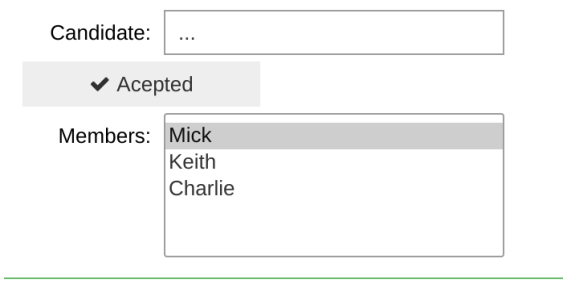
ipywidgets : Un scénario

Soit un widget composé ainsi :

```
import ipywidgets as widgets

wg = widgets.VBox(
    [
        widgets.Text(value="...", description="Candidate:"),
        widgets.Button(description="Acepted", icon="check"),
        widgets.Select(options=["Mick", "Keith", "Charlie"], description="Members:")
    ]
)
```

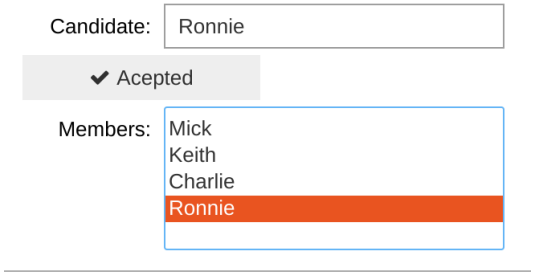
qui s'affiche comme cela :



ipywidgets : Un scénario (suite)

On souhaite rajouter une interaction correspondant au scénario suivant :

- Après avoir saisi un nouveau nom dans **Candidate**
- Et après avoir appuyé le bouton **Accepted**
  - => Le nouveau nom s'affiche dans la sélection



Une première implémentation du scénario

```
def _add_member(*args):
    opts = list(wg.children[2].options)
    wg.children[2].options = opts + [wg.children[0].value]
```

Cette implémentation n'est pas très lisible. On peut l'améliorer un peu :

```


CANDIDATE = 0
ACCEPTED_BTN = 1
MEMBERS = 2


def _add_member(*args):
    opts = list(wg.children[MEMBERS].options)
    wg.children[MEMBERS].options = opts + [wg.children[CANDIDATE].value]
```

Une seconde implémentation du scénario

En créant une sous-classe de **VBox**

```
class HandyVBox(widgets.VBox):
    def __init__(self, offsets, **kw):
        super().__init__(**kw)
        self._offsets = offsets

    def __getattr__(self, attr):
        offs = self.__getattribute__("_offsets")
        if attr in offs:
            return self.children[offs[attr]]
        return self.__getattribute__(attr)


wg = HandyVBox(
    offsets=dict(candidate=0, accepted_btn=1, members=2),
    children=[
        widgets.Text(value="...", description="Candidate:"),
        widgets.Button(description="Acepted", icon="check"),
        widgets.Select(options=["Mick", "Keith", "Charlie"], description="Members:")
    ],
)
```

permettant s'accéder les widgets via des attributs

Une seconde implémentation du scénario (suite)

Ce qui permet une écriture plus propre de l'interaction :

```
def _add_member(*args):
    opts = list(wg.members.options)
    wg.members.options = opts + [wg.candidate.value]


wg.accepted_btn.on_click(_add_member)
```

Malheureusement, aucune des deux implémentation n'est satisfaisante dans le contexte du typage statique

Une première tentative (naïve) de typage du widget:

```
def _add_member(*args: Any) -> None:
    opts = list(wg.children[MEMBERS].options)
    wg.children[MEMBERS].options = opts + [wg.children[CANDIDATE].value]


wg.children[ACCEPTED_BTN].on_click(_add_member)
```

... n'est pas une réussite :

```
(dlp2023) poli@pcpoli:~/DLP/dlp-python-typing-2023$ mypy --strict examples/untyped_widget_v1.py
examples/untyped_widget_v1.py:35: error: "DOMWidget" has no attribute "options"  [attr-defined]
examples/untyped_widget_v1.py:36: error: "DOMWidget" has no attribute "options"  [attr-defined]
examples/untyped_widget_v1.py:38: error: "DOMWidget" has no attribute "on_click"  [attr-defined]
Found 3 errors in 1 file (checked 1 source file)
(dlp2023) poli@pcpoli:~/DLP/dlp-python-typing-2023$
```

et le remède non plus :

```
def _add_member(*args: Any) -> None:
    opts = list(cast(widgets.Select, wg.children[MEMBERS]).options)
    cast(widgets.Select, wg.children[MEMBERS]).options = opts + [
        wg.children[CANDIDATE].value
    ]


cast(widgets.Button, wg.children[ACCEPTED_BTN]).on_click(_add_member)
```

La solution retenue

La solution précédente n'est pas satisfaisante car :

- elle détériore la lisibilité du code
- elle ne fiabilise pas le code non plus car les **cast()** sont manuels et l'erreur est humaine ...

On voudrait une solution qui permette de conserver le type des widgets contenus dans des attributs typés.

La solution retenue

Elle corrige les failles de la représentation précédente et elle passe par la définition d'un conteneur dédié :

```
class TypedVBox(widgets.VBox):
    Typed: type

    def __init__(self, children: Sequence[widgets.DOMWidget], **kw: Any) -> None:
        super().__init__(children, **kw)
        self.child = self.Typed()
        for i, (attr, typ) in enumerate(self.child.__annotations__.items()):
            assert isinstance(children[i], typ)
            setattr(self.child, attr, children[i])
```

ce conteneur est destiné à un usage général, non spécifique à notre exemple, et se spécialise facilement. Il suffit de définir une sous-classe dotée juste d'une classe imbriquée nommée **Typed** :

```
class MyTypedVBox(TypedVBox):
    class Typed:
        candidate: widgets.Text
        accepted_btn: widgets.Button
        members: widgets.Select
```

La solution retenue

Après avoir créé la sous-classe **MyTypedVBox** (dédiée à notre exemple) il ne reste plus qu'à l'instancier.

```
wg = MyTypedVBox(
    [
        widgets.Text(value="...", description="Candidate:"),
        widgets.Button(description="Acepted", icon="check"),
        widgets.Select(options=["Mick", "Keith", "Charlie"], description="Members:"),
    ]
)
```

L'interaction souhaitée s'écrit maintenant ainsi :

```
def _add_member(*args: Any) -> None:
    opts = list(wg.child.members.options)
    wg.child.members.options = opts + [wg.child.candidate.value]


wg.child.accepted_btn.on_click(_add_member)
```

et elle passe avec succès les vérifications en mode strict.

Intégration continue

Pour l'instant nous avons choisi d'effectuer les contrôles à 2 niveaux :

- coté client via **pre-commit**
- coté serveur via github/actions

Les 2 niveaux ne sont pas forcément redondants car coté serveur les contrôles se font dans un environnement vierge, potentiellement en avance par rapport à environnement de travail du développeur

Etapes pour adopter le type checking pour un code existant

- Si l'existant est très grand choisir une partie de taille raisonnable pour commencer
- Définir une même config mypy pour tous (.mypy.ini etc.)
- Ignorer les erreurs en provenance des modules provisoirement le cas echéant (option ignore_errors dans .mypy.ini)
- Gérer les erreurs induites par les dépendances :
  - Importer les packages xxx-stubs s'ils existent
  - utiliser `# type: ignore` si l'impact est faible
  - écrire les stubs manquants dans le cas contraire
- Annoter en priorité vos modules largement importés
- Reprendre depuis le début pour un autre sous-ensemble le cas echéant
- Annoter systématiquement le nouveau code
- Activer le type checking au niveau du CI et/ou pre-commit
- Passer si possible en mode strict, éventuellement en désactivant dans un premier temps les options les plus exigeantes

NB: La documentation de mypy préconise si besoin l'utilisation de générateurs de stubs, annotations etc. mais nous n'avons pas eu recours à ces solutions.

Bilan

Les avantages

- Fiabilite
- Lisibilité
- Aide à la documentation

Les inconvénients

- normes en pleine évolution
- stubs pas toujours disponibles

NB: Les inconvénients sont mineurs et inhérents à toute technologie nouvelle. Aucun des défis présenté précédemment ne correspond à une vraie situation de blocage. On a toujours le choix de baisser ponctuellement ses exigences quant elles sont trop difficiles à satisfaire et conserver les avantages du type checking pour les autres parties du code.

Merci !

Questions ?