27

Annotations et typage statique

Python est un langage fondamentalement dynamique. Toutes les variables manipulées par un programme peuvent, sur le papier, être passées en argument de n'importe quelle fonction ou méthode. Le contrôle est alors assuré pendant l'exécution par les exceptions : c'est le style de programmation EAFP (*Easier to Ask for Forgiveness than Permission*).

Dès les premières lignes de cet ouvrage, nous avons fait le choix de tirer profit du PEP 526 sur les annotations à des fins de documentation, pour clarifier les variables manipulées dans les exemples de code. C'est le premier cas d'utilisation des annotations : **documenter le code**, fournir des indications supplémentaires pour assister à la fois le rédacteur du code et l'utilisateur final. On notera notamment les différentes possibilités fournies pour annoter une même variable :

```
angle: float = 3.14                    radians = float
angle: "radians" = 3.14                angle: radians = 3.14
```

L'annotation `float` est « vraie ». Au fond, elle n'apporte pas grand-chose de plus que ce qui est déjà lisible dans le code : une nouvelle variable qui prend la valeur 3.14 est probablement de type flottant. Une annotation peut contenir n'importe quel élément de syntaxe Python valide, notamment une chaîne de caractères. Annoter la variable `angle` avec la chaîne de caractères `"radians"` est plus utile au développeur que l'annotation `float`, et se substitue alors à un commentaire du type « valeur de l'angle en radians ». Écrire `radians = float` permet de combiner les deux approches : le « vrai » (angle est de type `radians`, donc `float`) et la sémantique (on manipule une valeur d'angle en radians).

27.1. L'outil Mypy

Aujourd'hui, les géants du logiciel dépensent des fortunes pour produire des outils capables d'analyser le code, les annotations fournies et de **rechercher les incohérences avant l'exécution du code**. On appelle cette discipline l'*analyse statique* : on analyse le code non pas de manière dynamique à l'exécution, mais de manière statique avant de lancer le code [1].

1. Dans les langages compilés, c'est une étape qui a lieu en général avant la compilation.

À ce jour, les outils les plus répandus sont Mypy https://mypy.readthedocs.io (Dropbox) et Pyright https://github.com/Microsoft/pyright (Microsoft). Ces deux outils sont facilement intégrables dans les éditeurs de code classiques (VS Code, Vim, Sublime Text, Emacs, etc.)

L'idée est de pouvoir signaler au développeur les incohérences au moment où il écrit le code. L'objectif est de permettre au développeur, à l'image d'un correcteur orthographique, de corriger ces petites erreurs sans avoir à lancer le programme, lequel pourrait ne pas passer par les lignes problématiques.

```
radians = float
angle = 3.14
# plus loin...
angle = "radians"  # <<= l'éditeur devrait soulever une incohérence!
```

L'outil Mypy, en ligne de commande, permet de détecter cette incohérence. Les éditeurs de texte comme VS Code vont lancer Mypy (ou Pyright) en tâche de fond, et surligner les incohérences détectées au fur et à mesure (Figure 27.1).

```
$ mypy typing_01.py
typing_01.py:6: error: Incompatible types in assignment (expression has type "str",
variable has type "float")
Found 1 error in 1 file (checked 1 source file)
[1]    915758 exit 1     mypy typing_01.py
```

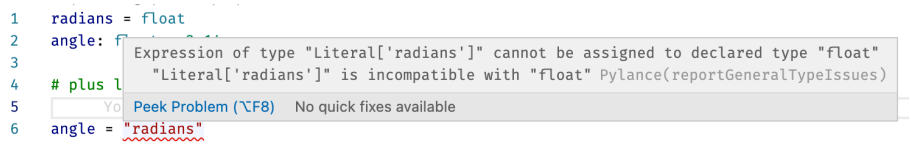

Figure 27.1 – L'éditeur VS Code intègre les résultats du programme Mypy dans son interface.

✧ Bonnes pratiques

Une troisième utilisation des annotations de type est de permettre aux éditeurs de texte de fournir des informations pour la **complétion automatique de code** (Figure 27.2). Sur la capture d'écran, l'éditeur comprend que la variable `m` est de type `Math` : il propose alors dans la liste de complétion l'ensemble des méthodes associées au type `Math`. Comme la méthode `Math.pi()` renvoie un flottant, alors la complétion propose l'ensemble des méthodes associées au type `float`.

27.2. L'annotation des fonctions

Les annotations de type en Python sont par définition facultatives. Par conséquent, les outils comme Mypy ne vont vérifier que les types des fonctions annotées et de leur résultat. Ceci permet d'ajouter des annotations de types à du code existant de manière progressive, en commençant par les modules les plus centraux et en terminant par les fonctionnalités de plus haut niveau. Cette particularité permet notamment de ne pas *typer* (ajouter des annotations de type) des fonctions dont les choix de conception compliquent cette mise en œuvre, sans qu'il soit possible de reprendre ce code dans l'immédiat.

Ainsi, une fonction non annotée ne sera pas vérifiée :

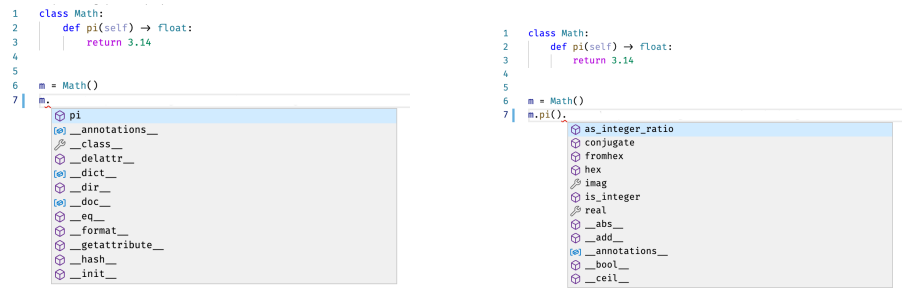

FIGURE 27.2 – L'éditeur VS Code utilise les annotations de type pour proposer une complétion intelligente.

```
def function():
    return 1 + ""  # pas d'erreur détectée

def function() -> None:
    return 1 + ""  # soulève une erreur
```

```
typing_03.py:2: error: Unsupported operand types for + ("int" and "str")
```

✧ Bonnes pratiques

L'annotation des arguments avec valeurs par défaut se fait sur le modèle suivant :

```
def distance(x1: float, y1: float, x2: float = 0, y2: float = 0) -> float:
    ...
```

✧ Bonnes pratiques

Les arguments de fonction `args` et `kwargs` (☞ p. 17, § 1.8) peuvent être annotés. Dans l'exemple ci-dessous, l'instruction `reveal_type` est comprise par Mypy pour aider le développeur ponctuellement mais devra être enlevée avant d'exécuter le code :

```
def fonction(*args: int, **kwargs: float):
    reveal_type(args[0])
    reveal_type(kwargs["facteur"])

typing_04.py:2: note: Revealed type is 'builtins.int*'
typing_04.py:3: note: Revealed type is 'builtins.float*'
```

27.3. Le module typing

Le module `typing` propose un certain nombre de types particuliers couramment utilisés.

Certains éléments du module `typing` sont parfois absents dans des versions antérieures de Python. Si tel est le cas, on peut alors les importer depuis une bibliothèque tierces nommée `typing_extensions`, qui recode ces fonctionnalités dans d'anciennes versions du langage.

Le type `Any` est un type qui est compatible avec n'importe quel type. Il est possible d'assigner une valeur de n'importe quel type à une variable de type `Any`, et d'appeler n'importe quelle méthode dessus.

```
from typing import Any

a: Any = None
a = 1
a = "hello"
a = a.avance()
```

Le type `Optional` permet de dire que la variable peut valoir `None`. Cette annotation permet de rattraper la plupart des erreurs de programmation détectables par analyse statique :

```
class Maybe:
    def maybe_none(self, x: int) -> int:
        if x > 0:
            return x

def fonction(a: Maybe: x: int = 1) -> int:
    return a.maybe_none(x) + 1
```

Ici, Mypy relève une erreur de typage sur la méthode `Maybe.maybe_none()` : il est facile d'oublier que l'instruction `return` est située dans un bloc `if`. Le cas `else` (omis ici) sous-entend donc que la méthode ne renvoie rien (`return None`).

```
typing_05.py:5: error: Missing return statement
```

Le type `Optional[int]` permet alors de préciser que la méthode renvoie soit `None`, soit un entier. Cette fois, c'est la ligne dans la fonction située plus loin qui cause une erreur. Il est probable que, dans tous les cas déjà rencontrés, `maybe_none()` a toujours renvoyé un entier. Cependant, l'utilisation des annotations par Mypy rappelle qu'il convient de traiter le cas où la valeur renvoyée est `None`.

```
from typing import Optional

class Maybe:
    def maybe_none(self, x: int) -> Optional[int]:
        if x > 0:
            return x
        return None

def fonction(a: Maybe, x: int = 1) -> int:
    return a.maybe_none(x) + 1
```

```
typing_05.py:13: error: Unsupported operand types for + ("None" and "int")
typing_05.py:13: note: Left operand is of type "Optional[int]"
```

On pourra par exemple corriger le code avec une exception :

```
def fonction(a: Maybe, x: int = 1) -> int:
    if res := a.maybe_none(x) is None:
        raise ValueError("maybe_none() a renvoyé None")
    return res + 1
```

Le type `Union` fait référence à une variable qui peut avoir plusieurs formats différents, p. ex. un entier ou un flottant, une liste ou une chaîne de caractères. On énumère alors toutes les possibilités de type que peut prendre la variable.

```
from datetime import datetime
from numbers import Number
from typing import Union

import pandas as pd

def make_timestamp(value: Union[Number, str, datetime, pd.Timestamp]) -> pd.Timestamp:
    """Convertit la valeur en entrée en timestamp Pandas.

    >>> make_timestamp("2020-12-25")
    Timestamp('2020-12-25 00:00:00')
    >>> make_timestamp(1608854400)
    Timestamp('2020-12-25 00:00:00')
    """
    if isinstance(value, str) or isinstance(value, datetime):
        return pd.Timestamp(value)
    if isinstance(value, Number):
        return pd.Timestamp(value, unit="s")
    return value
```

Le type `Union` peut également être utilisé avec l'opérateur `|` :

```
variable: Union[None, str, list[str]] = None
variable = None | str | list[str] = None  # notation équivalente, plus concise
```

27.4. Les types paramétrés

Les types paramétrés font référence à des structures de données qui dépendent d'un autre type. Une liste, ou un ensemble par exemple, est liée au type des éléments qui les composent. Pour un dictionnaire, on précisera le type de la clé en premier et le type des valeurs en second.

```
l: list[int] = [1, 3, 5, 3]
s: set[str] = {"un", "trois", "cinq"}
d: dict[int, str] = {1: "un", 3: "trois", 5: "cinq"}
```

Pour les tuples, on peut préciser un type pour tous les éléments d'un tuple de longueur inconnue, ou un type par valeur du tuple.

```
t1: tuple[int, ...] = (1, 3, 5)  # uniquement des entiers, longueur inconnue
t2: tuple[int, int, str] = (1, 3, "cinq")  # longueur fixe
```

✧ Bonnes pratiques

Le type `Optional[T]` est équivalent à `Union[None, T]`. Si un type `Union` plus complexe est optionnel, les deux notations `Union[None, int, str]` et `Optional[Union[int, str]]` sont équivalentes.

Bien qu'il n'y ait pas de vrais arguments pour préférer une des annotations à l'autre dans tous les cas, la première peut permettre de limiter le nombre de crochets pour améliorer la lisibilité.

Attention !

Si le type `Union` offre de la souplesse, il peut à l'inverse devenir contraignant pour son ambiguïté, son manque de précision sur la valeur annotée.

```
personne: dict[str, Union[str, int]] = {'prenom': 'Jean', 'age': 18}
majorite: bool = personne['age'] >= 18

typing_06.py:4: error: Unsupported operand types for >= ("str" and "int")
typing_06.py:4: note: Left operand is of type "Union[str, int]"
```

Dans ce cas particulier, on pourra préférer une autre structure de données, comme le `Namedtuple` ou la `dataclass`, ou encore utiliser le type `TypedDict` :

```
from typing import TypedDict

class Personne(TypedDict):
    prenom: str
    age: int

personne: Personne = {'prenom': 'Jean', 'age': 18}
majorite: bool = personne['age'] >= 18
```

Les types ABC permettent d'être le plus générique possible sur les types des variables d'entrée d'une fonction. La philosophie pour parvenir à typer rapidement et efficacement un programme consiste à être :

— le plus générique possible sur les paramètres d'entrée,
— le plus spécifique possible sur les paramètres de sortie.

Si une variable de sortie est définie de manière générale, il est plus difficile de connaître à l'avance les services qu'elle offre. Inversement, si une variable d'entrée est définie de manière spécifique, il devient difficile de passer une variable qui propose pourtant les mêmes services.

```
from typing import Iterable

def nonzero(seq: list[int]) -> list[int]:
    return list(elt for elt in seq if elt == 0)
```

```
n = nonzero({0, 1, 3, 5})
# error: Argument 1 to "nonzero" has incompatible type "set[int]";
# expected "lIst[int]"
```

Cette première option qui ne manipule que des listes est probablement trop restrictive : il est possible de passer des ensembles en entrée de la fonction sans perte de généralité, pourtant l'analyse statique échoue.

```
def nonzero(seq: Iterable[int]) -> Iterable[int]:
    return list(elt for elt in seq if elt == 0)

n = nonzero({0, 1, 3, 5})  # ok
n.append("sept")
# error: "Iterable[int]" has no attribute "append"
```

Cette deuxième option qui ne manipule que des structures génériques `Iterable` est également trop restrictive : l'analyse statique échoue sur l'appel à `.append()` qui fonctionne bien puisque le type de retour est une liste.

```
def nonzero(seq: Iterable[int]) -> list[int]:
    return list(elt for elt in seq if elt == 0)

n = nonzero({0, 1, 3, 5})
n.append("sept")
# error: Argument 1 to "append" of "list" has incompatible type "str";
# expected "int"

n.append(7)
# Success: no issues found in 1 source file
```

La troisième option est la meilleure : il suffit pour la variable `seq` de pouvoir itérer dessus ; mais la fonction renvoie bien une liste.

Tous les ABC sont ainsi disponibles : `Iterable[T]`, `Iterator[T]`, `Sequence[T]`, `Hashable`, `Mapping[K, V]`, etc. Dans la plupart des cas, une fonction génératrice (avec le mot-clé `yield`) pourra être typée avec `Iterator[YieldType]`. Pour une coroutine, on pourra utiliser le type `Generator[YieldType, SendType, ReturnType]`.

```
def stringify(seq: Iterable[int]) -> Iterator[str]:  # ou Generator[str, None, None]
    for elt in seq:
        yield str(elt)
```

Enfin, les fonctions d'ordre supérieur sont annotées avec l'ABC `Callable`. C'est la manière correcte de typer des fonctions, à préférer à la notation fléchée, courante dans les langages fonctionnels de la famille ML, que nous avons utilisée dans le chapitre 12 mais qui n'est pas comprise par Mypy : les types des paramètres sont passés sous forme de liste, et le deuxième argument est le type de retour.

Enfin, on utilise ici les fonctions du module `operator` pour faire appel aux fonctions associées aux opérateurs infixes `+`, `*` et `-`.

```
import operator
```

```
def sort_results(
    a: int, b: int,
    # au chapitre 12, on aurait écrit Iterable[int * int -> int]
    fonctions: Iterable[Callable[[int, int], int]]
) -> list[int]:
    return sorted(f(a, b) for f in fonctions)

sort_results(2, 1, [operator.add, operator.mul, operator.sub])
# [1, 2, 3]
```

Le type `TypeGuard` (PEP 647) permet de spécifier un type plus précisément dans des branches de code particulières. Supposons qu'on utilise un service web qui renvoie des objets JSON qui peuvent être d'un type `Temperature` ou d'un type `Pression`.

On peut définir les deux types, et définir le type JSON comme un type union :

```
from typing import Literal, TypedDict, TypeGuard

class Temperature(TypedDict):
    type: Literal["temperature"]
    valeur: int

class Pression(TypedDict):
    type: Literal["pression"]
    valeur: float

type JSON = Temperature | Pression

def is_temperature(field: JSON) -> TypeGuard[Temperature]:
    return field["type"] == "temperature"

def traitement(field: JSON) -> None:
    if is_temperature(field):
        x = field["valeur"]  # field est de type Temperature, x est de type int
    else:
        ...
```

27.5. Les types variables

Un type variable permet de modéliser « n'importe quel type », mais en lui donnant un nom : il permet ainsi de lier des types entre eux. La fonction `sort_results` par exemple pourrait être utile de manière plus générique, sans se limiter aux entiers. On souhaite ici que les types de `a` et de `b` correspondent aux types en entrée des fonctions.

Pour cela, on peut définir un type variable `T` :

```
from typing import TypeVar

T = TypeVar("T")
```

```
def sort_results(
    a: T, b: T,
    fonctions: Iterable[Callable[[T, T], int]]
) -> list[int]:
    return sorted(f(a, b) for f in fonctions)

sort_results(2, 1, [operator.add, operator.mul, operator.sub])
sort_results("un", "deux", [lambda a, b: len(a + b)])
# Success: no issues found in 1 source file
```

Il est possible de *contraindre* des types variables, c'est-à-dire d'énumérer les types qui peuvent convenir à la variable annotée `T`. À la différence d'un type `Union`, le type contraint fixe le type une fois pour toutes :

```
T = TypeVar("T", int, str)

def ajoute(a: T, b: T) -> T:
    return a + b

ajoute(1, 2)  # ok: int, int -> int
ajoute("un", "deux")  # ok: str, str -> str
ajoute(1, "deux")  # erreur: int, str
```

Depuis la version 3.12, il est possible de simplifier la notation sans avoir à importer `TypeVar` grâce à la syntaxe de paramètres de type (PEP 695).

```
def sort_results[T](
    a: T, b: T, fonctions: Iterable[Callable[[T, T], int]]
) -> list[int]:
    return sorted(f(a, b) for f in fonctions)

def ajoute[T: (str, int)](a: T, b: T) -> T:
    return a + b
```

Le PEP 695 introduit également le mot-clé `type` pour définir des alias de types (et simplifier les notations). On peut ainsi créer un type `Point` de la sorte :

```
type Point = tuple[float, float]
```

27.6. Les types génériques

Il est possible de créer ses propres types génériques, c'est-à-dire des types dépendant d'une variable d'un type encore inconnu au moment de l'analyse statique. La classe générique héritera alors de `Generic[T]`, où `T` est un type générique défini *a priori*. Ainsi, dans l'exemple suivant, on peut typer le décorateur `prefixe` à l'aide du type générique `T` :

```
from typing import Any, Generic

T = TypeVar("T", int, str)
```

```
class prefixe(Generic[T]):  # avec la nouvelle syntaxe: class prefixe[T: (int, str)]
    """Décorateur inutile, mais suffisamment simple pour l'exemple.

    Ce décorateur jouet ajoute systématiquement la valeur en paramètre
    au résultat de la fonction décorée."""

    def __init__(self, elt: T) -> None:
        self.elt: T = elt

    def __call__(self, fun: Callable[..., T]) -> Callable[..., T]:
        def newfun(*args: Any, **kwargs: Any) -> T:
            return self.elt + fun(*args, **kwargs)
        return newfun

@prefixe(">>> ")
def resultat_1() -> str:
    return "un"  # renvoie ">>> un" à cause du décorateur

@prefixe(2)
def resultat_2() -> int:
    return 2  # renvoie 4 à cause du décorateur

@prefixe(">>> ")
def resultat_3() -> int:
    return 3   # ">>> " + 3 n'est pas une opération valide
# error: Argument 1 to "__call__" of "prefixe" has incompatible type
# "Callable[[], int]"; expected "Callable[..., str]"

reveal_type(prefixe(">>>"))
# note: Revealed type is 'typing_14.prefixe[builtins.str*]'

reveal_type(prefixe(2))
# note: Revealed type is 'typing_14.prefixe[builtins.int*]'

@prefixe(2.4)
def resultat_4() -> float:  # float n'est pas dans les arguments de T
    return 4.1
# error: Value of type variable "T" of "prefixe" cannot be "float"
```

Plutôt que d'utiliser un type variable qui nous contraint à ne manipuler qu'un type `int` ou `str`, il est possible d'être un peu plus général pour accepter, entre autres, le type `float` pour `resultat_4`. D'après le code du décorateur, plus précisément de la fonction `newfun`, tout type valide vis-à-vis de l'addition pourrait convenir ici.

On peut alors réécrire l'exemple à l'aide du type paramétré `Protocol`, une simple classe qui ne contient que des définitions de méthodes annotées : le code n'importe pas, on peut se contenter des points de suspension.

```
T = TypeVar("T", bound="Addable")  # ①

class Addable(Protocol[T]):
    def __add__(self: T, other: T) -> T:  # ②
        ...

class prefixe(Generic[T]):
    def __init__(self, elt: T) -> None:
        self.elt: T = elt

    def __call__(self, fun: Callable[..., T]) -> Callable[..., T]:
        def newfun(*args: Any, **kwargs: Any) -> T:
            return self.elt + fun(*args, **kwargs)
        return newfun

@prefixe(">>> ")
def resultat_3() -> int:  # ③
    return 3
# error: Argument 1 to "__call__" of "prefixe" has incompatible type
# "Callable[[], int]"; expected "Callable[..., str]"

@prefixe(2.4)
def resultat_4() -> float:  # ④
    return 4.1

class Exemple:
    def __add__(self, other: "Exemple") -> "Exemple":
        return Exemple()

@prefixe(Exemple())  # ⑤
def resultat_5() -> Exemple:
    return Exemple()
```

① On utilise un type variable *borné* (*bounded* en anglais) : `T` est alors n'importe quel sous-type de `Addable`, n'importe quel type qui propose l'addition.

② `Addable` définit l'opérateur addition : les deux arguments, `self` et `other`, et le type de retour sont les mêmes.

③ Mypy détecte que le type de retour de `resultat_3` n'est pas compatible avec le paramètre passé à `prefixe`.

④ On peut manipuler des flottants qui sont valides vis-à-vis du calcul de l'addition.

⑤ La classe `Exemple` propose aussi la méthode spéciale `__add__(self, other)` dans ses services.

27.7. Le typage des décorateurs

Le type `ParamSpec` (PEP 612) permet de reporter la signature d'une fonction quelle qu'elle soit dans la signature d'une autre fonction. Dans l'exemple ci-dessous, on veut typer correctement un décorateur, et s'assurer que la fonction décorée porte les mêmes annotations de variables.

```
from asyncio import sleep
from typing import Awaitable, Callable, ParamSpec, TypeVar

P = ParamSpec("P")
R = TypeVar("R")

def attendre_un_peu(fun: Callable[P, R]) -> Callable[P, Awaitable[R]]:
    async def fonction_async(*args: P.args, **kwargs: P.kwargs) -> R:
        await sleep(10)
        return fun(*args, **kwargs)

    return fonction_async

@attendre_un_peu  # avec le décorateur, le type de retour est Awaitable[int]
def takes_int_str(x: int, y: str) -> int:
    return x + 7

async def main() -> None:
    await takes_int_str(1, "A")  # correct
    c: int = takes_int_str(1, "A")  # "note: Maybe you forgot to use "await"?"
    await takes_int_str("B", 2)  # incorrect
```

27.8. Variance : covariance et contravariance

La variance est la discipline qui traite des relations de sous-typage. Python est un langage de programmation orienté objet, et des questions se posent quant aux relations d'héritage.

Reprenons notre classe `Polygone` et ajoutons-y deux méthodes :

— `simplify()` spécifie comment simplifier des polygones aux formes complexes pour réduire le nombre de points qui les composent tout en préservant au mieux leurs formes. On peut imaginer par exemple coder l'algorithme de Visvalingam[2] à cette fin ;

— `__lt__()` compare les aires des polygones.

```
class Polygone:

    def simplify(self) -> "Polygone":
        ...  # algorithme de Visvalingam

    def __lt__(self, other: "Polygone") -> bool:
        ...
```

2. https://en.wikipedia.org/wiki/Visvalingam-Whyatt_algorithm

Comment typer la méthode `simplify` *pour la classe* `Triangle` *?*

Le type de retour `Triangle` pourrait convenir.

Comment typer la méthode `simplify` *pour la classe* `Hexagone` *?*

Il n'y a aucun moyen de connaître à l'avance le type de retour ; tout dépend des spécificités de l'hexagone passé en entrée. Le plus sûr sera de spécifier un type de retour `Polygone`.

Une méthode `Triangle.__lt__(self, other: "Triangle") -> bool` *est-elle acceptable ?*

Non. Cette signature est trop restrictive par rapport à l'interface de `Polygone`, qui promet d'accepter n'importe quel type `Polygone`.

```
error: Argument 1 of "__lt__" is incompatible with supertype "Polygone";
supertype defines the argument type as "Polygone"
```

Les réponses à ces questions se formalisent avec trois qualificatifs :

- un *constructeur de type* **covariant** autorise le sous-typage dans le même sens que le type en paramètre. Ainsi, la méthode `simplify()` dans les sous-classes de `Polygone` peut renvoyer aussi bien un `Polygone` qu'un sous-type de celui-ci ;
- un constructeur de type **contravariant** (moins intuitif) autorise le sous-typage dans le sens opposé au type en paramètre. La méthode `__lt__(self, other)` peut accepter en paramètre un type `Polygone`, ou n'importe quel type plus général, par exemple `Union[Polygone, Cercle]` ;
- un constructeur de type **invariant** interdit tout sous-typage. C'est la solution la plus sûre d'un point de vue de la vérification des types, mais aussi la moins utilisable.

Le constructeur de type d'une fonction (ou d'une méthode) peut alors être :

- *covariant* par rapport au type de retour ;
- *contravariant* dans les types des paramètres d'entrée.

✧ Bonnes pratiques

Lors du typage d'une fonction, on gagne en utilisabilité en choisissant des types :

- les plus génériques possibles pour les arguments (dans le sens de la contravariance, du plus spécifique au plus générique) ;
- les plus spécifiques possibles pour le type de retour (dans le sens de la covariance).

Dans la fonction suivante :

```
def sorted_non_none(seq: Iterable[T]) -> list[T]:
    return sorted(elt for elt in seq if elt is not None)
```

Un type `list[T]` pour le paramètre d'entrée serait correct mais interdirait *de facto* l'usage d'ensembles ou de fonctions génératrices qui seraient pourtant acceptés par la fonction : il est préférable de typer `Iterable[T]`.

Un type `Iterable[T]` pour le paramètre de sortie serait correct mais interdirait alors d'appliquer une méthode applicable aux listes sur le résultat de la fonction : il est préférable d'afficher l'ensemble des fonctionnalités accessibles sur le type de retour avec le type `list[T]`.

Il est possible de construire des types covariants ou contravariants à l'aide des arguments `covariant` et `contravariant` du constructeur `TypeVar`. Les occasions de manipuler ces arguments en pratique sont plutôt rares. Le lecteur est invité à se référer au PEP 484 le cas échéant.

En quelques mots...

Les annotations de type permettent de détecter un grand nombre d'erreurs, souvent faciles à résoudre, avant d'exécuter le code. Ces annotations sont facultatives, mais il y a toutefois un effet de seuil dans un grand projet à partir duquel on ressent les bénéfices des annotations, et les maladresses ou erreurs commencent à être efficacement repérées. Plus les types d'entrée sont génériques, inclusifs et plus les types de sortie sont précis, prescriptifs, plus grande sera la plus-value apportée par l'analyse statique de code.

Un code mal annoté, ou annoté partiellement, reste exécutable, au même titre qu'un code où les types sont erronés. Une annotation difficile à appréhender peut être enlevée, ou remplacée par `Any`, dans l'attente de trouver une solution plus tard, à court, moyen ou très long terme, voire jamais. Une ligne de code peut aussi être marquée comme à ignorer par l'analyseur statique avec le commentaire `## type: ignore`

Enfin, les annotations permettent de réduire le volume de commentaires et de documentation pour en améliorer la lisibilité. Les informations des types sont alors placées au plus près des variables, là où l'œil recherche l'information. On ajoute souvent l'exécution d'un analyseur statique, comme Mypy, dans les outils de vérification de code, avant l'exécution des tests automatiques, pour surveiller la viabilité du code d'un projet et la qualité des modifications proposées par les développeurs tiers.

Est-ce que tout le monde devrait annoter son code ?

Non. Les utilisateurs du langage Python ont tous un profil différent, et tous ne sont pas sensibles à la logique des types.

Un programmeur débutant aura probablement déjà beaucoup à faire avec d'autres aspects du langage. Les annotations de type n'apporteront probablement guère plus qu'une complexité inutile. Un *data analyst* qui code quelques lignes de Pandas et Matplotlib dans un notebook ne verra aussi aucune plus-value à annoter son code : l'objectif de sa démarche étant d'arriver rapidement à des résultats ou à un prototype qui valide sa faisabilité.

En revanche, un code partagé, destiné à être réutilisé dans d'autres projets, par soi ou par d'autres utilisateurs, qui passent souvent peu de temps dans la documentation, gagne beaucoup à être annoté. Ces annotations pourront, au même titre que la documentation, être exploitées par les éditeurs de code, pour proposer de la complétion de code ou pour surligner des erreurs et mauvaises utilisations de la bibliothèque.

Pour aller plus loin

— Le site suivant propose de nombreux exercices de typage :
`https://github.com/laike9m/Python-Type-Challenges`