



<https://pa.dilla.fr/1c>

Chap. 4 – Mise au point de programmes

4.1 – Types

4.1.1 – Les types en Python

Chaque valeur manipulée par un programme Python est associée à un **type**, qui caractérise la nature de cette valeur.

ACTIVITÉ 1

La fonction `type` permet d'obtenir le type de la valeur passée en paramètre. **Utilise** cette fonction pour **déterminer** le type de `1`, `3.14`, `True`, `"abc"`, `None`, `(1, 2)`, `[1, 2, 3]`, `{1, 2, 3}`, `{'a': 1, 'b': 2}`.

2. **Détermine** le type de `{}`, d'une fonction et d'une classe.

CORRECTION

valeur	type	description
1	int	nombre entiers
3.14	float	nombre décimaux
True	bool	booléens
"abc"	str	chaînes de caractères



None	NoneType	valeur indéfinie
-	-	-
(1, 2)	tuple	n-uplets
[1, 2, 3]	list	tableaux
{1, 2, 3}	set	ensembles
{'a': 1, 'b': 2}	dict	dictionnaires

En Python, la gestion des types est qualifiée de **dynamique** : c'est au moment de l'exécution du programme, lors de l'interprétation de chaque opération de base, que l'interprète Python vérifie la concordance entre les opérations et les types des valeurs utilisées.

4.1.2 – Annoter les variables et les fonctions

Il est **indispensable** lors de la définition d'une fonction d'avoir en tête les types attendus pour

- les paramètres et
- l'éventuel type du résultat.

Pour la définition d'une interface, cette information est cruciale et permet d'éviter autant que possible la mauvaise utilisation d'un module.

Python accepte l'annotation des **variables** et des **fonctions**.

REMARQUE

Ces annotations sont facultatives dans ce langage mais obligatoires dans d'autres. Elles ont pour rôle :

- de documenter le code (utile pour toute relecture)



— de permettre une vérification *statique* (avant l'exécution) des types par des programmes externes.

Exemple

```
[ ]: # annotation des variables
x: int = 42
```

Exemple

```
[ ]: # Annotation des fonctions

def contient_doubleton(t: list) -> bool:
    # annotation du paramètre : tableau (list)
    # et de la sortie : valeur booléenne (bool)
    pass

def cree () -> list:
    # annotation de la sortie : tableau (list)
    pass

def contient(s: list, x: int) -> bool:
    # annotation des paramètres : tableau (list)
    # nb entier (int)
    # et de la sortie : valeur booléenne (bool)
    pass

def ajoute(s: list, x: int) -> None:
    # annotation des paramètres : tableau (list)
    # nb entier (int)
    # et explicitement aucune sortie : indéfinie (None)
    pass
```

4.1.3 – Types nommés et types paramétrés

REMARQUE

En Python, les informations de types pour les **valeurs structurées** (n-uplets, tableaux, dictionnaires, etc.) restent **très superficielles**.

Un couple d'entier (comme (1, 2)) et un triplet mixte (comme (1,



"abc", False)) sont tous les deux le type `tuple` alors qu'ils n'ont quand même rien à voir...

Pour préciser les types de ces valeurs structurées, il faut utiliser le module `typing`. Il définit de nouvelles versions des types de base : les types `Tuple`, `List`, `Set`, `Dict`.

Ces nouvelles versions acceptent un ou plusieurs paramètres en fonctions du ou des types de leurs composants.

type	description
<code>Tuple[int, bool]</code>	couple d'un entier et d'un booléen
<code>List[int]</code>	tableau d'entiers
<code>Set[str]</code>	ensemble de chaînes de caractères
<code>Dict[str, int]</code>	dictionnaire dont les clés sont des chaînes de caractères et les valeurs des entiers

Exemple

```
[ ]: from typing import List

def cree () -> List[int] :
    # annotation de la sortie : tableau d'entiers (List[int])
    pass
```


4.2 – Tester un programme

4.2.1 – Tester la correction d'une fonction

Pour vérifier qu'une fonction fait bien ce qu'elle est sensée faire il faut effectuer des tests.



Afin d'aider à la mise au point des programmes, on peut annoter les fonctions Python avec des **types**, qui décrivent la nature des arguments et des résultats. Bien qu'il ne s'agisse là que d'une forme de documentation supplémentaire, ignorée par l'interprète Python, des outils externes permettent une **vérification statique** de ces types, c'est-à-dire une vérification **avant** que le programme ne soit exécuté. La mise au point des programmes passe également par une phase de test rigoureuse, qui s'assure de la correction mais également des performances. Le test de fonctions Python peut avantageusement se contruire autour de l'instruction `assert`.


ACTIVITÉ 2

Implémenter la classe `Intervalle` définissant l'intervalle `a..b` (noté aussi `[a;b]`).

- Ajouter** une méthode `est_vide()` vérifiant si l'intervalle est vide (un intervalle tel que $b \leq a$ est considéré comme vide).
- Vérifier** que la méthode `est_vide()` est correcte.

CORRECTION

Définissons la classe `Intervalle` qui définit un intervalle d'extrémité `self.a` et `self.b` :

```

class Intervalle:
    def __init__(self, debut, fin):
        """Intervalle d'extrémité [a ; b]"""
        self.a = debut
        self.b = fin

    et une méthode est_vide qui renvoie une valeur booléenne associée au
    prédicat //intervalle est vide :

    ...
    def est_vide(self):
        """Est ce que l'intervalle est vide?"""
        return self.b < self.a

    Pour tester la fonction est_vide on pourra vérifier que l'exécution du
    programme ci-dessous affiche bien False puis True.

    # premier test
    
```

Programmation défensive pour les méthodes

```

class C:
    def __init__(self, x, y):
        if not (...invariant...):
            raise ValueError('...explication...')
        self.x = x
        self.y = y

    ...

    def deplace(self):
        if ...:
            self.x += 1
            self.y += 1
        assert (...invariant...)
    
```

Exemple

Lorsque la vérification d'un invariant commence à être complexe, on peut
 déporter cette vérification dans une méthode spécifique.

```

class C:
    ...
    def valide(self):
        ...vérifie l'invariant...
        ...et lève une exception si besoin...
    
```



```
mon_inter = Intervalle(5, 12)
print( mon_inter.est_vide() )

# deuxième test (écrit en une ligne)
print( Intervalle(5,3).est_vide() )
```

Le module `doctest` propose une façon pratique d'intégrer les tests et les résultats attendus **directement dans la méthode** (ou la fonction) concernée. Un appel à la fonction `testmod()` effectue l'ensemble des tests et vérifie si le résultats escompté est affiché. La synthèse des tests effectués est affichée dans l'interprète Python.

Pour utiliser cet outil, il faut :

1. importer la fonction `testmod` du module `doctest`
2. modifier la documentation des fonctions et méthodes
3. exécuter la fonction `testmod()`
4. étudier l'interprète pour vérifier la bonne exécution des tests.

Exemple

Ainsi pour la méthode `est_vide()` de l'activité précédente, on écrira :

```
from doctest import testmod

class Intervalle
...
    def est_vide(self):
        """Est ce que l'intervalle est vide?
        >>> mon_inter = Intervalle(5,12)
        >>> mon_inter.est_vide()
        False
```



4.3 – Invariants de structure

Il n'est pas rare que les attributs d'une classe satisfassent des **invariants** (propriétés qui restent vraies tout au long de l'exécution du programme).

Exemple

Voici quelques exemples possibles d'invariants de structure :

- un attribut représentant un mois de l'année a une valeur comprise entre 1 et 12;
- un attribut contient un tableau d'entiers et représente le numéro de sécurité social. La taille du tableau doit être de 13;
- un attribut contient une mesure d'angle qui doit être comprise entre 0 et 360°;
- un attribut contient un tableau qui doit être trié en permanence;
- deux attributs `x` et `y` représentent une position sur une grille $N \times N$ et ils doivent donc respecter les inégalités : $0 \leq x < N$ et $0 \leq y < N$
- etc.

REMARQUE

Concernant la programmation orientée objet. Le principe d'encapsulation de la programmation objet permet d'imaginer maintenir ces invariants. Il suffit que le constructeur de la classe les garantisse puis que les méthodes qui modifient les attributs maintiennent ces invariants.

Exemple

Programmation défensive pour le constructeur :

Après la correction, on souhaite le plus souvent vérifier les **performances** de nos programmes.

La théorie permet de prédire les performances (ça s'appelle la **complexité** et nous l'étudierons dans l'année).

Une façon simple et efficace est de mesurer le temps d'exécution d'un programme. Pour cela, nous allons utiliser la fonction `time()` de la bibliothèque `time` qui renvoie le nombre de secondes écoulées depuis un instant de référence (1 janvier 1970 à minuit, démarrage de l'ordinateur, etc.).

```
[ ] : from time import time

print(time())
print(time())
```

La valeur affichée ne nous intéresse pas, c'est la **différence** entre deux valeurs qui nous indiquera la **durée** d'exécution !

Exemple

```
[ ] : tab = tableau_aléatoire(10_000, -1_000, 1_000)

t0 = time()
tri(tab)
print(time() - t0)
```

Plutôt que de mesurer les performances d'un seul appel, il est préférable d'essayer de faire varier les entrées, dans le but de relier la taille de ces entrées avec la mesure du temps d'exécution.

Exemple

```
[ ] : for k in range(10, 16):
    n = 2 ** k
    tab = tableau_aléatoire(n, -100, 100)
    t = time()
    tri(tab)
    print(n, time() - t)
```

10

4.2.2 – Tester la correction d'un programme

ACTIVITÉ 3

Écrire une fonction `tri1` qui trie un tableau d'entiers, en place, par ordre croissant.

On cherche maintenant à tester la fonction `tri1`.

2. **Proposer** une fonction `test()` qui prend en argument un tableau `t`, appelle la fonction `tri1` sur ce tableau puis vérifie que le tableau `t` est bien trié.

– implémenter une fonction de test naïve

– vérifier que le tableau avant tri et après tri contient les mêmes éléments, et pour chaque élément le même nombre d'occurrence.

7

On peut remarquer que :

```
>>> Intervalle(5, 3).est_vide()
True
"""
return self.b < self.a

testmod()
```

- les tests sont écrits directement dans la documentation de la fonction (`docstring`)
- les instructions à interpréter sont précédées de trois chevrons `>>>`
- les résultats attendus sont écrits directement
- il suffit d'écrire l'instruction `testmod()` pour lancer les tests.



```
[ ]: # exemple de fonction de tri qui doit échouer :
from typing import List

def tri(t: List[int]) -> None:
    """Efface tout (et donc c'est trié!)"""
    t.clear()

def tri(t: List[int]) -> None:
    """Supprime les doublons mais trie"""
    tab = []
    for x in t:
        if x not in tab:
            tab.append(x)
    tab.sort()
    t.clear()
    for x in tab:
        t.append(x)
```

CORRECTION

```
[ ]: def occurrences(t):
    """renvoie le dictionnaire des occurrences de t

    Args:
        t (list): tableau en entrée
    """
    d = {}
    for x in t:
        if x in d:
            d[x] += 1
        else:
            d[x] = 1
    return d

def identiques(d1, d2):
    """deux dictionnaires identiques

    Args:
        d1 (dict)
        d2 (dict)
    """
    for x in d1:
        assert x in d2
        assert d1[x] == d2[x]
    for x in d2:
        assert x in d1
        assert d2[x] == d1[x]

def test(t):
    """teste la fonction tri sur le tableau t

    Args:
        t (list): tableau à tester
    """
    occ = occurrences(t)
    tri(t)
    for i in range(0, len(t) - 1):
        assert t[i] <= t[i+1]
```



```
identiques(occ, occurrences(t))
```

**ACTIVITÉ 4**

Maintenant que la fonction test est correcte, on peut passer à des tests un peu plus ambitieux.

1. À l'aide de la fonction `randint` de la bibliothèque `random`, crée une fonction `tableau_aleatoire(n: int, a: int, b: int) -> List[int]` qui renvoie un tableau de `n` éléments pris aléatoirement dans l'intervalle `a..b`.
2. Utilise la fonction précédente pour effectuer 100 tests effectués sur des tableaux de différentes tailles et dont les valeurs sont prises dans des intervalles variables.

CORRECTION

```
[ ]: from random import randint
from typing import List

def tri(t):
    """fonction de tri correcte"""
    t.sort()

def tableau_aleatoire(n: int, a: int, b: int) -> List[int]:
    return [randint(a,b) for _ in range(n)]

for n in range(100):
    # [0,0,...,0]
    test(tableau_aleatoire(n,0,0))
    # tableau avec bcp de doublons
    test(tableau_aleatoire(n, -n//4, n//4))
    # tableau de grande amplitude
    test(tableau_aleatoire(n, -10*n, 10*10))
```

4.2.3 – Tester les performances