Dans ce chapitre, on revient sur quelques bonnes habitudes que l'on peut prendre lorsqu'on souhaite développer un logiciel ambitieux. Nous allons notamment aborder le typage des données, les tests, et les invariants de structure.

1 Typage des données

Chaque donnée en Python (mais également dans tous les langages de programmation) est affublée d'un type. Rappelons-les :

Type	Exemple	Description
`int`	`3`	Entiers relatifs
`float`	`2.18`	Nombres flottants
`bool`	`True`	Booléens
`NoneType`	`None`	Type indéfini
`str`	`"coucou"`	Texte
`tuple`	`(1,2)`	n-uplet
`list`	`[1,2]`	Tableau
`set`	`{1,2}`	Ensemble
`dict`	`{"toto":1, "tata":2}`	Dictionnaire

Certaines opérations ont un sens différent en fonction du types des données sur lesquelles elles sont exécutée. Ainsi l'opération

```
a+b
```

Pourra être une addition si `a` et `b` sont des entiers, une concaténation si `a` et `b` sont des tableaux, voire pourra générer une `TypeError` si `a` et `b` sont de types incompatibles, par exemple `int` et `list`.

Dans un long programme, il peut être souhaitable afin de garder en mémoire les types de chaque variable d'annoter celles-ci, c'est à dire de faire apparaître leur type dans le code même. On procède comme dans les exemple suivants :

— `n: int = 52`

— `x: float = 23.1`

— `t: list = [1,2,3]`

Remarque : Ces annotations sont purement indicatives et n'influent en rien le code. En particulier, la commande :

— `n: int = 3.14145`

ne générera aucune erreur. Python ne vérifie en effet pas la cohérence des type *a priori*. Dans beaucoup d'autre langage, cette vérification aura en revanche bien lieu et un typage incohérents des données empêchera le programme d'être compilé : on parle de **vérification statique** des types.

Pour les fonctions, il convient d'annoter les types des arguments mais aussi du résultat. On le fait comme dans l'exemple suivant :

```
def minimax(t: list) -> tuple:
    mini: int, maxi: int = t[0], t[0]
    for x in t:
        if x > maxi:
            maxi = x
        if x < mini:
            mini = x
    return mini, maxi
```

On peut ainsi voir d'un simple coup d'oeil à la lecture du code que la fonction `minimax` prend un argument un tableau et renvoie un n-uplet.

Remarque : Les fonctions ne retournant aucun résultat renvoie en réalité la valeur `None`. Ainsi une fonction travaillant par effet de bord uniquement pourra se repérer par une déclaration de la forme :

— `def f(x: int) -> NoneType:`

Pour les types construits (`tuple`, `list`, `set` et `dict`), qui stockent un collection de données, il est possible de préciser le ou les types de celles-ci.

Exemple	Description
`tuple[int, bool]`	couple entier / booléen
`list[float]`	tableau de flottants
`set[str]`	ensemble de textes
`dict[int, str]`	dict. à clées entières, valeurs textuelles

Remarque : On retrouve une convention qu'on essaie de suivre en python : stoker toujours des données de même type dans les tableaux, ensembles et dictionnaires. Le type de chaque coordonnée est à préciser pour les n-uplets.

Reprenons l'exemple de fonction précédent :

```
def minimax(t: list[int]) -> tuple[int, int]:
    mini: int, maxi: int = t[0], t[0]
    for x in t:
        if x > maxi:
            maxi = x
        if x < mini:
            mini = x
    return mini, maxi
```

Cette fois, on précise que l'argument est un tableau d'entiers et que le résultat est un couple d'entiers.

Il est aussi possible lorsqu'un type paramétré revient fréquemment dans un programme de donner un nom particulier à ce type. On parle alors d'**alias de type**.

1. `Ensemble = list[int]`
2. `Ensemble = list[list[int]]`

D'une manière similaire, le type d'un objet d'une certaine classe est le nom de cette classe.

```
class Chrono:
    """classe simulant un chronometre"""
    def __init__(self: Chrono) -> Chrono:
        self.temps = 0

    def avance(self: Chrono) -> NoneType:
        ...

    def __str__(self: Chrono) -> str:
        ...

c: Chrono = Chrono()
```

2 Tester un programme

On parle de test d'une fonction ou d'un programme lorsqu'on écrit une autre fonction ou un autre programme dont le but est de vérifier que le premier fonctionne correctement.

Considérons par exemple une fonction de tri par effet de bord à tester :

— `tri(t: list[int]) -> NoneType`

Pour déterminer si celle-ci fonctionne correctement, il nous faut écrire une fonction permettant de déterminer si le résultat de celle-ci correspond bien à la version triée de son argument.

```
def occurences(t):
    """renvoie le dictionnaire des occurrences de t"""
    d = {}
    for x in t:
        if x in d:
            d[x] += 1
        else:
            d[x] = 1
    return d

def identiques(d1, d2):
    """verifie que les dictionnaires d1 et d2 sont identiques"""
    for x in d1:
        assert x in d2, "cle non-commune"
        assert d1[x] == d2[x], "occurrences distinctes"
    for x in d2:
        assert x in d1, "cle non-commune"
        assert d1[x] == d2[x], "occurrences distinctes"

def test(t):
    """verifie que l'appel tri(t) trie effectivement t"""
    occ1 = occurrences(t)
    tri(t)
    for i in range(len(t)-1):
       assert t[i]<t[i+1], "tableau non trie !"
    occ2 = occurrences(t)
    assert identiques(occ1, occ2), "valeurs differentes !"
```

Grâce à la fonction `test(t)`, on peut vérifier la correction de notre fonction sur des tableaux divers. On parle de jeu de tests.

```
from random import randint

def tableau_aleatoire(n, a, b):
    """renvoie un tableau de n valeurs tirees au hasard entre a et b"""
    return [randint(a, b)] for i in range (n)]

def test_tri():
    """test de la fonction tri"""
    for n in range(100):
        test(tableau_aleatoire(n, 0, 0))
        test(tableau_aleatoire(n, -n//4, n//4))
        test(tableau_aleatoire(n, -10*n, 10*n))
```

La fonction `time()` du module `time` nous permet de mesurer le temps d'exécution d'une fonction. Reprenons l'exemple de notre fonction de tri.

```
from time import time

def temps_tri(t):
    """renvoie le temps de tri sur le tableau t"""
    debut = time()
    tri(t)
    return time()-debut
```

Pour avoir une idée des performances de notre fonction, il va nous falloir évaluer don temps d'exécution moyen pour des tableaux de tailles croissantes.

```
def perf_tri():
    d = {}
    for k in range(10, 16):
        d[k] = 0
        for i in range(10):
            t = tableau_aleatoire(2**k, -100, 100)
            d[k] += temps_tri(t)
        d[k] = d[k]/10
    return perf
```

3 Invariants de structure

En POO, on parle d'invariant de structure pour désigner une propriété qui doit toujours être vérifiée par certains attributs d'une classe. Par exemple :

— dans une classe `Date`, l'attribut `mois` est un entier compris entre 1 et 12 ;

— dans une classe `Fraction`, l'attribut `denom` est toujours un entier strictement positif ;

De tels invariants peuvent être :

1. vérifiés dans le constructeur si on y passe des valeurs d'attributs en argument ;
2. vérifiés voire maintenus dans chaque fonction modifiant ces derniers.

On pourra utiliser pour vérifier que les invariants sont bien respectés des `assert` à la fin de chaque fonctions modifiant les attributs. Cela peut également être fait à l'aide d'une méthode `valide(self)`.

```
from math import gcd

class Fraction:
    def __init__(self, n, d):
        self.num = n
        self.denom = d

    def valide(self):
        """test des invariants"""
        assert type(self.num) == int
        assert type(self.denom) == int
        assert self.denom>0
        assert gcd(self.num, self.denom) == 1

    def simplifie(self):
        """maintenance de l'irreductibilite"""
        if self.num != 0:
            d = gcd(self.num, self.denom)
            self.num = self.num // d
            self.denom = self.denom //d
```