Dans ce chapitre, on revient plus en détail sur les bonnes pratiques de programmation à mettre en oeuvre dans le cadre du développement d'un logiciel.

Nous allons aborder le typage des données, les tests, et les invariants de structure.

1 Typage des données

1.1 Types simples, types construits, classes

Chaque donnée en Python (mais aussi dans les autres 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

En programmation orientée objet, **toute instance d'une classe a pour type le nom de la classe**.

Certaines opérations ont un sens différent en fonction du type des données sur lesquelles elles sont exécutée, on parle de **polymorphisme**. Ainsi la commande :

```
a+b
```

pourra résulter en 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`.

1.2 Annoter les variables et les fonctions

Dans un programme Python, il est souhaitable de préciser le type de donnée contenu dans chaque variable à sa déclaration. On parle d'**annotations de type** (*type hints*).

```
n: int = 52
x: float = 23.1
t: list = [1,2,3]
```

Ici, l'annotation nous indique que les variables `n`, `x` et `t` contiennent respectivement des données de type `int`, `float` et `list`.

De même, il convient d'annoter les types des arguments et du résultat des fonctions.

```
def minimax(t: list) -> tuple:
    """Renvoie le couple (min, max) des valeurs du tableau."""
    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
```

Ici, l'annotation nous indique que la fonction `minimax` prend en argument un tableau et renvoie un n-uplet.

Remarque : une fonction sans `return` renvoie par défaut la valeur `None`. On l'annote avec le type `None` (type et valeur sont confondues).

```
def f(x: int) -> None:
```

1.3 Vérification des types

Les annotations de types en Python sont, du point de vue de l'interpréteur, purement indicatives et n'influent en rien l'exécution du code. En particulier, la commande suivante ne génère aucune erreur.

```
n: int = 3.14145
```

Python ne vérifie en effet pas la cohérence les annotations de types et les types des données effectivement stockées dans les variables.

Il existe des outils externes permettant d'effectuer de telles vérifications comme `mypy` ou `pytype`. Certains IDE comme Pycharm ou Visual Studio peuvent aussi se charger de cette vérification.

La gestion des types varie d'un langage de programmation à l'autre.

Ainsi, Python analyse les types des variables au fur et à mesure de l'exécution du programme et c'est notamment pour cela qu'il est possible d'en changer. On dit que Python est un langage à **typage dynamique**.

Dans beaucoup d'autres langages de programmation, le typage a lieu à la compilation, avant l'exécution : on parle de **typage statique**. Celui-ci peut se faire par des annotations de type obligatoires (C) ou automatiquement via un mécanisme appelé *inférence de type* (Caml).

1.4 Le module `typing`

Python dispose d'un module `typing` permettant d'annoter les types des variables de manière plus précise.

Pour les objets de types construits (`tuple`, `list`, `set` et `dict`), stockant une collection de données, il permet de préciser 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

Remarques :

— Pour utiliser de telles annotations, il faut importer le module `typing`.

— Contrairement à leurs équivalents simples, les types paramétrés prennent une majuscule.

Reprenons l'exemple de la fonction `minimax` :

```
from typing import List, Tuple

def minimax(t: List[int]) -> Tuple[int, int]:
    """Renvoie le couple (min, max) des valeurs du tableau."""
    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**.

```
from typing import List, Tuple, Dict

Ensemble = List[int]
Vecteur = Tuple[float, float]
Bulletin = Dict[str, int]
```

Si on veut indiquer qu'un type paramétré peut accepter plusieurs types de valeurs, on peut utiliser une **variable de type** `T` :

```python
from typing import List, Tuple, Dict, TypeVar

T = TypeVar("T")
EnsembleGenerique = List[T]
```

2 Tests des programmes

2.1 Tests : définition et classification

On parle de **test** pour désigner un programme dont l'objectif est de vérifier le bon fonctionnement d'un autre programme (ou fonction) à travers un certain nombre de **cas à tester** (*test case*) prédéfinis.

Lors du développement d'un logiciel, le passage régulier par des **phases de test** est essentielle. Certains postes au sein d'une équipe informatique tels que testeur QA (*Quality Assurance*), analiste QA ou intégrateur y sont d'ailleurs dédiés.

Il existe de nombreuses sortes de tests, faisons-en un petit tour :

On peut classer les différents tests un tests selon *la nature des objets testés et / ou le contexte du test* :

— **test unitaire :** test d'une petite unité de code – typiquement une fonction, réalisé généralement directement par l'équipe de développement ;
— **test d'intégration :** test d'une partie du logiciel – typiquement un module, réalisée par l'équipe de test ;
— **test système :** test du logiciel entier selon ces spécifications réalisée par des l'équipe de test ;
— **test d'acceptation :** test du logiciel entier selon ces spécifications réalisée par le client.

On peut aussi les classer selon les propriétés testées :

— **test fonctionnel (correction) :** on vérifie que l'objet testé fait ce qu'il est supposé faire ;
— **test de performance (complexité) :** on quantifie les performances de l'objet testé comme son temps d'exécution ou son utilisation mémoire ;
— **test d'intrusion :** test de la sécurité du logiciel ;
— **test utilisateur :** on vérifie qu'un utilisateur peut utliser le logiciel.

Nous allons voir des exemples de tests unitaires, fonctionnels et de performances.

2.2 Exemple de test fonctionnel unitaire

Considérons une fonction de tri sur place à tester :

```python
tri(t: List[int]) -> None
```

Pour vérifier si l'exécution de cette fonction est correcte sur un certain tableau `t`, on doit vérifier deux choses :

1. le tableau est trié après l'exécution ;
2. le tableau comporte les mêmes valeurs avant et après l'exécution.

On commence par écrire des fonctions auxiliaires permettant ces vérifications.

```python
def est_trie(t: List[T]) -> bool:
    """Teste si le tableau t est trié."""
    for i in range(len(t)-1):
        if t[i] > t[i+1]:
            return False
    return True

def occurences(t: List[T]) -> Dict[T, int]:
    """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
```

Grâce à ces deux fonctions, on peut écrire une fonction de test.

```
def test(t: List[T]) -> None:
    """Vérifie que l'appel tri(t) trie effectivement t."""
    occ1 = occurrences(t)
    tri(t)
    occ2 = occurrences(t)
    assert est_trie(t), "tableau non-trié !"
    assert occ1 == occ2, "valeurs différentes !"
```

La procédure de test étant définie, on doit générer les cas de tests, une fonction générant un tableau aléatoire va donc être utile :

```
def tableau_aleatoire(n: int, a: int, b: int) -> List[int]:
    """Renvoie un tableau de n entiers aléatoire entre a et b."""
    return [randint(a, b) for i in range (n)]
```

On peut finalement écrire le programme de test en entier.

```
# Fichier test_tri.py
# Test fonctionnel de la fonction tri du module mon_module

from typing import List, TypeVar
from random import randint
from mon_module import tri

# Définition des fonctions vues plus haut
...

# Tests
for n in range(100):
    test(tableau_aleatoire(n, 0, 0))          # t remplis de 0
    test(tableau_aleatoire(n, -n//4, n//4))   # t avec doublons
    test(tableau_aleatoire(n, -10*n, 10*n))   # t avec sans doublons
```

À noter : En cas de bon fonctionnement de la fonction de tri, ce programme n'affichera rien du tout, les `assert` ne faisant que lever une `AssertionError` dans le cas où elles ne sont pas respectées.

2.3 Exemple de test de performance unitaire

La fonction `perf_counter` du module `time` permettent de mesurer le temps d'exécution d'une fonction. Cette fonction est une version plus précise de la fonction `time` existant depuis la version 3.3 de Python.

```
def perf(t: List[T]) -> float:
    """Renvoie le temps de tri du tableau t en secondes."""
    debut = perf_counter()
    tri(t)
    fin = perf_counter()
    return fin-debut
```

Comme pour les tests fonctionnels, on veut appeler cette fonction sur des entrées de tailles variées afin de mesurer la réaction de notre fonction à des tailles de plus en plus élevées.

Ces données peuvent être affichées de manière brutes dans un terminal, stockées dans un `log` ou encore compilées sous forme de graphique.

```
# Fichier test_tri.py
# Test de performances de la fonction tri du module mon_module

# Import des modules nécessaires
...
import matplotlib.pyplot as plt

# Définition des fonctions vues plus haut
...
```

```python
# Tests
X: List[int] = []
Y: List[float] = []
for k in range(10):
    n = 1000*k
    print(f"Test du tri d'un tableau de taille {n}")
    t: List[int] = tableau_aleatoire(n, -n//4, n//4)
    temps: float = perf(t)
    X.append(n)
    Y.append(temps)

plt.plot(X, Y)
plt.show()
```

Dans l'exemple ci-dessus, on génère une courbe à l'aide du module matplolib.

Voici des exemples de courbes obtenues pour différentes fonctions de tri avec les mêmes valeurs de tailles (on aurait pu pousser plus pour `t.sort()`) :

Remarque : On visualise avec ses courbes un résultat bien connu : les algorithmes de tri par insertion et par sélection sont **quadratiques**.

3 Invariants de structure

En programmation orientée objet, 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 ;

En appliquant les principes de l'**encapsulation** et de la **programmation défensive**, de tels invariants peuvent être garantis :

1. par vérification ;
2. par construction ;

par le constructeur de la classe et les méthodes modifiant des attributs.

On pourra utiliser pour vérifier que les invariants sont bien respectés des commandes `assert`.

```python
from math import gcd

class Fraction:
    """Classe représentant des fractions.
    attributs :  num (numérateur), denom (dénominateur)
    invariant : n > 0
    invariant : d et n sont premiers entre eux
    """
    def __init__(self, n:int , d:int):
        """Constructeur"""
        self.num = n
        self.denom = d
        assert n > 0, "le numérateur doit être strictement positif"
        assert gcd(n, d) == 1, "num et denom doivent être premiers entre eux"
```

Si les vérifications alourdissent trop le code, on peut écrire une méthode `valide(self)` effectuant la vérification.

```python
from math import gcd

class Fraction:
    """Classe représentant des fractions.
    attributs :  num (numérateur), denom (dénominateur)
```

```
    invariant : n > 0
    invariant : d et n sont premiers entre eux
    """
    def __init__(self, n:int , d:int) -> None:
        """Constructeur"""
        self.num = n
        self.denom = d
        self.valide()

    def valide(self) -> None:
        assert n > 0, "le numérateur doit être strictement positif"
        assert gcd(n, d) == 1, "num et denom doivent être premiers entre eux"
```

Bien que plus difficile à mettre en place, il est plus intéressant de garantir les invariants par construction, c'est à dire de faire en sorte dans le code même qu'il soit impossible que le constructeur ou toute autre méthode modifiant un attribut aboutisse à un objets ne vérifiant pas les invariants.

Prenons pour exemple une classe `Chrono` représentant des durées sous la forme de trois attributs `h`, `m`, `s` et disposant d'une méthode permettant d'ajouter (pas enlever) un certain nombre de secondes.

On veut maintenir les invariants suivants :

— `0 <= h`;

— `0 <= h <= 59`;

— `0 <= h <= 59`;

Le caractère positif des durées peut être simplement vérifié. Si en revanche on dépasse 59 pour les secondes ou les minutes, on augmentera le l'attribut `minute` ou `heure` en conséquence.

```
class Chrono:
    """Classe représentant un chronomètre."""
    def __init__(self, h:int , m:int, s:int) -> None:
        """Constructeur."""
        assert s >= 0 and m >= 0 and h >= 0, "durées positives"
        self.h = h
        self.m = m
        self.s = s
        self.maintenir()

    def ajoute(self, s: int) -> None:
        """Ajoute s secondes au chronomètre."""
        assert s >= 0, "durées positives"
        self.s = self.s + s
        self.maintenir()

    def maintenir(self) -> None:
        """Maintient les secondes et les minutes entre 0 et 59."""
        self.h = self.h + self.m//60 + self.s//3600
        self.m = (self.m + self.s//60)%60
        self.s = self.s%60
```