PGR. Mise au point de programmes

Ce chapitre est une introduction aux bons réflexes à prendre pour rendre un programme lisible et surtout gérer les éventuelles erreurs, du programmeur lui-même mais aussi de l'utilisateur.
On considère qu'environ 60% d'un programme est consacré … à la gestion des erreurs !

Trois grand thèmes seront abordés ici :

- Le **typage**, qui permet la détection précoce d'incohérences dans un programme.
- Les **annotations** qui rendent un programme plus lisible.
- Le **test systématique** et rigoureux de différentes fonctions composant un programme.

I/ Types en programmation

Chaque valeur manipulée par un programme est associée à un **type** qui **caractérise la nature de cette valeur**. On a ainsi un type *int* pour les nombres entiers.

Types principaux en langage Python

Valeur	Type	Description
	Types de base	
1	**int**	nombres entiers
3.2	**float**	nombre décimaux
True	**bool**	booléens
« cde » ou 'cde'	**string (str)**	chaîne de caractères
None	**NoneType**	valeur indéfinie
	Types composés	
('e' , 3)	**tuple**	*n*-uplets
[1 , 'yop' , False]	**list**	tableaux
{ 'a' : 1 , 'b' : 2 , 'c' : 3 }	**dict**	dictionnaires

A noter : En langage C, il n'existe pas de types composés (il faut les créer soi-même) mais on peut manipuler des adresses mémoire à l'aide de pointeurs.

Les types permettent de caractériser les opérandes (*) ou paramètres qui sont ou non acceptables pour certaines opérations. Le langage Python est par nature faiblement typé (on parle de **typage dynamique** (*)) mais l'utilisation de valeurs qui seraient par nature incompatibles avec une opération donnée lèvera une exception *TypeError* qui est accompagnée d'un message d'information sur les types qui ne conviennent.

Exemple d'une addition incompatible avec les types choisis :

```
1 # Tentative d'additionner un entier
2 # avec une liste
3 print(1 + [2,3])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-1-158a11e1fa40> in <module>
      1 # Tentative d'additionner un entier
      2 # avec une liste
----> 3 print(1 + [2,3])

TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

Typage dynamique : il s'agit de laisser le programme décider à quel type correspond une valeur lors de son exécution. A l'inverse, on parlera de **typage statique** lorsque ce rôle revient au programme : c'est au développeur de l'indiquer.

Opérandes : il s'agit des valeurs auxquelles on applique une opération arithmétique ou logique. En d'autres termes, il s'agit des termes composant une opération.

Dans le cas d'opérateur comme l'addition qui peuvent avoir plusieurs significations (on parle de *surcharge* de cet opérateur), on peut d'ailleurs la redéfinir dans une classe, par exemple pour additionner deux vecteurs.

Exemples	Type des opérandes	Effet	Résultat
1 + 2	int	addition	3
1.2 + 3.4	float	addition	4.6
True + True (à éviter)	bool	addition	2
« abc » + « ade »	string (str)	concaténation	« abcade »
(1 , 2) + (1 , 3)	tuple	concaténation	(1 , 2 , 1 , 3)
[1 , 2] + [4 , 6]	list	concaténation	[1 , 2 , 4 , 6]

Rappel : Types construits, *listes*, *dictionnaires* et *n-uplets*.
Lien vers la documentation des structures de données en Python :
https://docs.python.org/3/tutorial/datastructures.html

II/ Annotation de variables et de fonctions

Pour rendre un code plus lisible, on ne saurait que conseiller des **donner** des **noms explicites** aux **variables** et aux **fonctions** et de les **documenter** si le besoin s'en fait sentir.

Il peut être opportun – en plus d'adopter les conseils ci-dessus – d'annoter certaines variables et fonctions.

Quelques exemples d'annotations ici :

```python
# Annotation d'une variable
# On spécifie son type
age : int = 42

# Annotation d'une fonction
# On spécifie le type du paramètre
# et celui de la valeur de retour
def prix_paye(age : int) -> float :
    if age < 18 :
        return 6.30

    return 9.50
```

```python
# Annotation d'une méthode
# Le type de `self`est la classe
class Chrono :
    def texte(self : Chrono) -> str :
        return "Le temps passé est"
```

A noter : les annotations sont à titre documentaire et n'imposent aucun typage !

Il existe des vérificateurs de types pour Python. Un outil historique pour la vérification statique des types est *mypy* que l'on trouve à cette adresse : http://mypy-lang.org/
Cet outil peut être ajouté à toute installation standard de Python et fournit un programme que l'on peut exécuter sur un ensemble de fichiers *.py* pour réaliser la vérification, et, le cas échéant, obtenir la liste des erreurs.

Plusieurs grandes compagnies ont produit leur propre vérificateur, qu'il s'agisse d'un mypy (*pytype* pour Google, *pyre* pour Facebook) ou d'outils intégrés dans des environnements de développement (EDI) comme par l'exemple l'éditeur *Pycharm* ou le *Visual Studio Code*.

On peut aussi indiquer le rôle d'une fonction (ou d'un programme) en encadrant les explications sur **plusieurs lignes** par des **triples guillemets**.
On les place au-dessus de la fonction en question ou au début du programme.

Voici un exemple :

```
""" Chiffrement du message : conversion en entier de chaque lettre
du message et de la clé et application du "ou exclusif" (a^b),
l'instruction i % long_cle permet de faire en sorte
que la clé soit de la même taille que le message."""

def chiffre(message_d,cle_d) :
    message_dec_code= []
    long_cle = len(cle_d)

    # Parcours du message à coder
    for i in range(len(message)) :
        message_dec_code.append(message_d[i] ^ cle_d[i%long_cle])

    return message_dec_code
```

III/ Tests de programmes

Il est primordial de tester les programmes avant de les mettre à disposition. Il y a plusieurs types d'erreurs possibles :

- Des **erreurs de frappes** (étourderies) rapidement identifiée au lancement du script.
- Des **erreurs à l'utilisation** (mauvaises données entrées).
- Des **erreurs d'algorithmes** qui peuvent être difficiles à trouver du fait qu'elles n'aboutissent pas toujours à une erreur d'exécution mais à des résultats incohérents.

On peut intercepter certaines erreurs lors de la saisie de valeurs à l'aide **d'instructions conditionnelles**.
Voici un exemple d'un programme où l'on attend l'âge d'une personne en argument : la valeur donnée doit être un entier positif (que l'on peut limiter à 150 ans par exemple).

```
1  # Détermine le prix d'un billet de cinéma
2  def prix_billet(age : str) -> str :
3      # Vérification du type
4      if not age.isnumeric() or int(age) != float(age):
5          return "L'âge doit être un entier"
6
7      i_age = int(age)
8
9      # Vérification des bornes
10     if i_age < 0 or i_age > 150 :
11         return "L'âge doit être compris entre 0 et 150 ans"
12
13     if i_age < 18 :
14         return "Le billet coûte 6.50 euros"
15     else :
16         return "Le billet coûte 9 euros"
17
18 s_age = input("Entrez un âge : ")
19 print(prix_billet(s_age))
20
21 # Jeu de tests
22 # s_age = "19" # Attendu : "Le billet coûte 9 euros"
23 # s_age = "yop" # Attendu : "L'âge doit être un entier"
24 # s_age = "6.0"  # Attendu : "L'âge doit être un entier"
25 # s_age = "159" # Attendu : "L'âge doit être compris entre 0 et 150 ans"
```

```
Entrez un âge : 159
L'âge doit être compris entre 0 et 150 ans
```

L'instruction ***assert*** *condition à respecter, texte à afficher le cas échéant* permet de remplacer avantageusement une instruction conditionnelle. En effet, elle possède deux avantages :

- Cette instruction est réservée au traitement d'erreurs alors que c'est n'est pas la fonction première d'une instruction conditionnelle.
- Elle permet de stopper de programme à la ligne de l'erreur avec un message explicite que le programme peut ajouter.

```python
# Détermine le prix d'un billet de cinéma
def prix_billet(age : str) -> str :
    # Vérification du type
    assert age.isnumeric() and int(age) == float(age), "L'âge doit être un entier"

    i_age = int(age)

    # Vérification des bornes
    assert i_age >= 0 and i_age <= 150, "L'âge doit être compris entre 0 et 150 ans"

    if i_age < 18 :
        return "Le billet coûte 6.50 euros"
    else :
        return "Le billet coûte 9 euros"

s_age = input("Entrez un âge : ")
print(prix_billet(s_age))
```

Si on essaie de rentrer « a » comme âge, une exception est levée et le programme s'arrête tout en indiquant l'endroit où s'est produite l'erreur.

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-12-bae613ad959e> in <module>
     15
     16 s_age = input("Entrez un âge : ")
---> 17 print(prix_billet(s_age))

<ipython-input-12-bae613ad959e> in prix_billet(age)
      2 def prix_billet(age : str) -> str :
      3     # Vérification du type
----> 4     assert age.isnumeric() and int(age) == float(age), "L'âge doit être un entier"
      5
      6     i_age = int(age)

AssertionError: L'âge doit être un entier
```

Une autre utilité de l'instruction ***assert*** est de tester des fonctions.

Voici un exemple avec une fonction ***tri()*** (que l'on n'a pas obligatoirement écrite d'ailleurs) qui se propose de trier une liste par ordre croissant.

```python
def test(tab : list) -> NoneType :
    # Tri du tableau
    tri(tab)
    # Test de ce tri
    for i in range(0, len(tab) - 1) :
        assert tab[i] <= tab[i+1], "Tableau non trié"
```

Lors de l'appel de la fonction ***test(tab)***, une erreur sera levée si le tri n'est pas conforme à l'attendu.

<u>Important</u> : contrairement à l'instruction conditionnelle, il faut écrire la condition qui DOIT ETRE VERIFIEE avec l'instruction *assert*.

On peut également tester de manière plus précise que la méthode ***isnumeric()*** à l'aide de la fonction ***type(variable)***. Combiné avec une assertion, on peut par exemple stopper un programme si un utilisateur n'a pas rentré une variable du bon type.

Un petit exemple très simple :

```
def donne_age(age) :
    assert type(age) == int, "Ce n'est pas un nombre entier"
    print("Vous avez ", age, " ans")


donne_age(10)  # Ok
donne_age('e') # Erreur !
```

```
Vous avez  10  ans

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_20268/1167909145.py in <module>
      5
      6 donne_age(10)  # Ok
----> 7 donne_age('e') # Erreur !

~\AppData\Local\Temp/ipykernel_20268/1167909145.py in donne_age(age)
      1 def donne_age(age) :
----> 2     assert type(age) == int, "Ce n'est pas un nombre entier"
      3     print("Vous avez ", age, " ans")
      4
```

Enfin, la fonction ***isinstance(value, (type_1, type_2 …))*** permet de vérifier si une variable (1er paramètre) est bien de l'un des types décrits dans le n-uplet (2ème paramètre) :

```
var_1 = 9
var_2 = "yop"
var_3 = [2,3]

# `var_1` est un entier : True
if isinstance(var_1, (int,float)) :
    print(f"{var_1} est soit un entier ou un nombre flottant\n")
# `var_2` est une chaîne de caractères : True
assert isinstance(var_2, (str)), f"{var_2} : Mauvais type\n"
# `var_3` n'est pas un dictionnaire : False
assert isinstance(var_3,(dict)), f"{var_3} : Mauvais type\n"
```

```
9 est soit un entier ou un nombre flottant


---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_8312/1108420057.py in <module>
      9 assert isinstance(var_2, (str)), f"{var_2} : Mauvais type\n"
     10 # `var_3` n'est pas un dictionnaire : False
---> 11 assert isinstance(var_3,(dict)), f"{var_3} : Mauvais type\n"

AssertionError: [2, 3] : Mauvais type
```