Python

Hadrien Bodin

Mines

28 octobre 2025

Table des matières

Pourquoi Python ?

La lisibilité :

- pas de délimitations begin end ;
- uniquement des indentations
- la lisibilité fait partir de l'ADN du langage
- typage dynamique

Pourquoi Python ?

La puissance du langage :

- types très puissants et flexibles
  - entiers non bornés, nombres complexes
  - listes, strings Unicode
  - tables de hash : dictionnaires et ensembles
  - langage orienté objet : définir ses propres types
- un grand nombre de librairies (qui s'interfacent facilement en C et Cpp)
- gestion automatique de la mémoire

Pourquoi Python ?

La compilation :

- Python est un langage interprété
- Un script s'effectue ligne par ligne
- Cela permet un usage interactif
- Retenir cependant qu'une précompilation existe

Pratiques à adopter

Un code doit être intelligible. Il faut se souvenir qu'un code est plus souvent lu qu'écrit.
Quelques principes à toujours appliquer :

- Keep it short and simple
- Nom des fonctions, variables etc explicite
- Commentaires SSI nécessaires
- Annotations de type

Règles de nommage

Un relecteur doit être capable de comprendre l'intérêt d'un objet à partir de son seul nom.

- On peut écrire des noms "longs"
- Choisir une unique langue pour la nomenclature

Le lecteur a un "contexte d'attention". Si une variable n'est utilisée que sur quelques lignes, on peut se permettre de la nommer de façon moins explicite.

Règles de nommage

Un relecteur doit être capable de comprendre l'intérêt d'un objet à partir de son seul nom.

- On peut écrire des noms "longs"
- Choisir une unique langue pour la nomenclature

Le lecteur a un "contexte d'attention". Si une variable n'est utilisée que sur quelques lignes, on peut se permettre de la nommer de façon moins explicite.

Règles de nommage

- On peut écrire des noms "longs"
- Choisir une unique langue pour la nomenclature

Les variables et les objets

- Variables et fonctions : minuscule + underscore
- Constantes : Tout en majuscule
- Classe : Majuscule + minscule

```
position_x, position_y, masse_1, masse_2 # des variables
NOMBRE_PLANETES # une constante
get_acceleration() # une fonction
Simulation() # une classe
```

Les commentaires

Bien penser à aérer le code (sauter des lignes quand nécessaire).
Commentaire pour ce qui est nécessaire : souvent les noms de variables suffisent.

Annotation de type

Il est de bonne pratique d'annoter les types des variables attendues et renvoyées par une fonction.

- Pour la lisibilité
- Pour le contrôle des erreurs

On déclare le type de chaque paramètre de la fonction, ainsi que le type de ce qui est retourné par la fonction.

Annotation de type

```
def get_item(my_list : list[float], pos : int = 2)
    -> float:
    return my_list[pos]
```

Un bon IDE

Configurer son IDE pour travailler proprement.

Les variables

En Python, une variable peut être vue comme une adresse vers un objet. Ce qui est derrière cette adresse peut changer. Cela n'a aucune importance pour l'interpréteur.

```
a = '1'
type(a)
    -> 'str'
a = int(a)
type(a)
    -> 'int'
```

La mutabilité

Un objet mutable est un objet qui peut-être modifié une fois créé.
Les entiers ne sont pas mutables

```
a = b = 1
a += 1
a is b
    -> False
```

a pointe initialement vers l'objet 1.
Comme cet objet n'est pas mutable, modifier la valeur de a correspond à créer un nouvel objet (2) et à faire pointer a vers ce nouvel objet.

La mutabilité

A contrario, les listes, par exemple, sont mutables

```
a = b = [1]
a.append(2)
a is b
    -> True
print(b)
    -> [1, 2]
```

Argument des fonctions

Il est parfois utile d'utiliser des arguments par fonction dans une fonction.
La valeur d'un tel argument est évalué à la création de la fonction.

Attention

Attention aux objets mutables en valeur par défaut

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
    -> [1]
print(f(2))
    -> [1, 2]
print(f(3))
    -> [1, 2, 3]
```

Argument des fonctions

- Les fonctions peuvent avoir des arguments obligatoires ou non.
- Les arguments peuvent être nommés ou positionnels.
- Les arguments non obligatoires ont des valeurs par défaut.
- A la définition, il faut d'abord déclarer les arguments obligatoires puis les autres.
- A l'appel, l'ordre des arguments nommés n'importe pas

Arguments obligatoires

```
def f(a, b):
    print(f"{a}_{b}")
f(1,2)
    -> 1_2
f(a=1, b=2)
    -> 1_2
f(b=2, a=1)
    -> 1_2
f(1)
    -> f() missing 1 required positional argument: 'b'
```

Argument des fonctions

- Dans un appel de fonction, les arguments nommés doivent suivre les arguments positionnés.
- L'ordre des arguments nommés (obligatoires ou non) n'importe pas.

Les *args et **kwds permettent de passer un nombre arbitraire d'arguments (réciproquement non nommés et nommés).

Gestion des erreurs

Vous avez pu le voir, il est commun d'avoir des erreurs dans notre code.
Soit car on les rencontre, soit car on souhaite les causer.
Il faut distinguer les erreurs de syntaxe et les erreurs d'exécution.
ex : indentation manquante vs division par zéro

Catcher une erreur

Parfois on est conscient qu'une erreur peut survenir et on ne souhaite pas faire crasher le programme si on la rencontre.
La syntaxe try, except sert à cela.
On essaie de faire quelquechose, en s'attendant à un échec.

Créer notre propre type d'objet

Une fois ces bases posées, il est possible de s'intéresser à la programmation orientée objet.
Pour décrire un objet complexe, il semble pertinent d'avoir besoin de recourir à une structure de donnée complexe.

Exemple

Quel type python choisir pour décrire un ou une élève ? Une classe ? Une école ?

Les classes

Pour définir de nouveau types, on utilise une classe.
La classe peut être vue comme un générateur d'objets.
Tous les objets issus de la même classe partagent les mêmes propriétés (mais pas forcément les mêmes valeurs).

Les classes

Les objets issus d'une classes contiennent :

- Des attributs
- Des méthodes

```
a = np.array([1])
a.shape
a.mean()
```

Créer une classe

Le décorateur à utiliser est **class**

```
class Eleve:
    age = 25
    def say_hello():
        print("Hello")

e = Eleve
e
    -> <class '__main__.Eleve'>
e = Eleve()
e
    -> <__main__.Eleve object at 0x0000019FE45C5010>
```

Ici on pointait dans un premier temps vers la classe, et ensuite vers un objet instancié par la classe.
Retenir que toutes les méthodes doivent à minima contenir l'argument self.

Accéder aux éléments de la classe

On peut accéder aux différents éléments de la classe.

```
e = Eleve()
e.say_hello()
    -> Hello
e.age
    -> 25
e.age = 30
e.age
    -> 30
e.note = 15
e.note
    -> 15
```

Attention, si changer un attribut ou en rajouter un à la volée est possible, cela est souvent déconseillé.
Il faut souvent considérer les attributs comme privés : Seule la classe elle même devrait avoir le droit de les modifier.

init

L'instanciation en appelant un objet classe crée un objet vide.
Il peut cependant être pertinent de créer des instances personnalisées correspondant à un état initial spécifique.
À cet effet, une classe peut définir une méthode spéciale nommée __init__(). L'instanciation de la classe appelle automatiquement cette méthode.

```
class Eleve:
    def __init__(self):
        self.age = 25
    def say_hello():
        print("Hello")
```

init

On comprend tout de suite qu'il est pertinent de donner des arguments à la méthode init.

```
class Eleve:
    def __init__(self, age, nom):
        self.age = age
        self.nom = nom
    def say_hello():
        print("Hello")

e = Eleve(25, "Hadrien")
```

Variables de classe

En général, les variables d'instance stockent des informations relatives à chaque instance alors que les variables de classe servent à stocker les attributs et méthodes communes à toutes les instances de la classe.

Attention à ne pas abuser des variables de classe.

```
class Eleve:
    ecole = 'Mines'          # class variable shared by all instances
    def __init__(self, nom):
        self.nom = nom     # instance variable unique to each instance

>>> d = Eleve('eleve_1')
>>> e = Eleve('eleve_2')
>>> d.ecole                          # shared by all dogs
'Mines'
>>> e.ecole                          # shared by all dogs
'Mines'
>>> d.nom                          # unique to d
'eleve_1'
>>> e.nom                          # unique to e
'eleve_2'
```

Variables de classe

Les objets mutables peuvent présenter des surprises

```
class Eleve:
    ecole = 'Mines'              # class variable shared by all
    notes = []
    def __init__(self, nom, note):
        self.nom = nom     # instance variable unique to eac
        self.notes.append(note)

e = Eleve('h', 15)
e.notes
    -> [15]
d = Eleve('i', 20)
e.notes
    -> [15, 20]
```

Il vaut donc mieux toujours utiliser des variables d'instances

repr

Pour une raison ou une autre, on peut avoir envie de print un objet.

```
print(d)
    -> <__main__.Eleve object at 0x000002721492041O>
```

Par défaut le résultat n'est pas très pertinent... Pour cela il existe la méthode __repr__.
Le retour de cette fonction est affiché par le print.

repr

```
class Eleve:
    def __init__(self, nom):
        self.nom = nom
    def __repr__(self):
        return f"eleve: {self.nom}"

e=Eleve("Hadrien")
print(e)
    -> Eleve: Hadrien
```

Exercice

Exercice

Créer une classe Polygone sans utiliser numpy :

- Un polygone est constitué d'un ensemble de Points (Classe à écrire)
- A sa création, on lui adjoint un nom. Une liste de coordonnées peut par ailleurs être renseignée (non obligatoire).
- Ecrire une méthode pour ajouter un point
- Ecrire une méthode pour supprimer le i-ème point
- Ecrire une méthode pour récupérer la position du i-ème point
- Ecrire une méthode pour translater le polygone le long d'un vecteur
- Un print de l'objet doit afficher, entre autres, le nombre de points, et le barycentre

Héritage

Si deux classes sont proches, on peut vouloir faire hériter une classe fille d'une classe mère :
On peut par exemple avoir une classe "Personne" qui contient des informations sur quelqu'un.
Une classe "Eleve" aurait donc toutes les raisons de partager des similarités avec la classe "Personne".

Héritage

```
class Personne:
    def __init__(self, age, nom):
        self.age = age
        self.nom = nom

    def say_hello():
        print('hello je suis un humain')

    def say_goodbye():
        print('goodbye')

class Eleve(Personne):
    def __init__(self, age, nom, ecole):
        self.ecole = ecole
        super().__init__(age, nom)

    def say_hello():
        print('hello je suis   tudiant  ')
```

Opérateurs

Les opérateurs classiques que l'on utilise en python peuvent être vus comme de simple décorateurs syntaxique.
Derrière leur exécution, se cache l'appel à une fonction.
Par exemple, sommer deux entiers revient à appeler une méthode de la Classe int qui prend un autre int en paramètre.

Opérateurs

On comprend donc qu'il peut être pertinent de redéclarer ces méthodes pour les classes que l'on construit.
Par exemple, il peut être pertinent de pouvoir sommer deux vecteurs.

Déclaration d'un opérateur

```
class My_int:
    def __init__(self, val):
        self.val = val
    def __add__(self, my_other_int):
        return GFG(self.val + my_other_int.val)
```

Itérateur

Si on a un objet de type Ecole, qui contient une liste d'Eleves, il serait souhaitable de pouvoir écrire :

```
ecole = Ecole()
for eleve in ecole :
```

Cela n'est possible que si la classe Ecole() définit un iterateur.

Définir un itérateur

Pour définir un itérateur, il peut être pertinent d'implémenter les méthodes suivantes :

- __iter__
- __next__
- __len__
- __getitem__

Iter

Cette fonction doit renvoyer un objet qui est itterable. Cet objet peut être un des attributs de la classe mère

Next

Cette fonction doit renvoyer le prochain objet que l'on souhaite récupérer.
Une fois tous les objets récupérés, cette fonction doit raise un StopIteration

len

Cette fonction doit renvoyer le nombre d'objets sur lesquels on itère

Cette fonction prend en paramètre une position et renvoie le n-ième itérable.