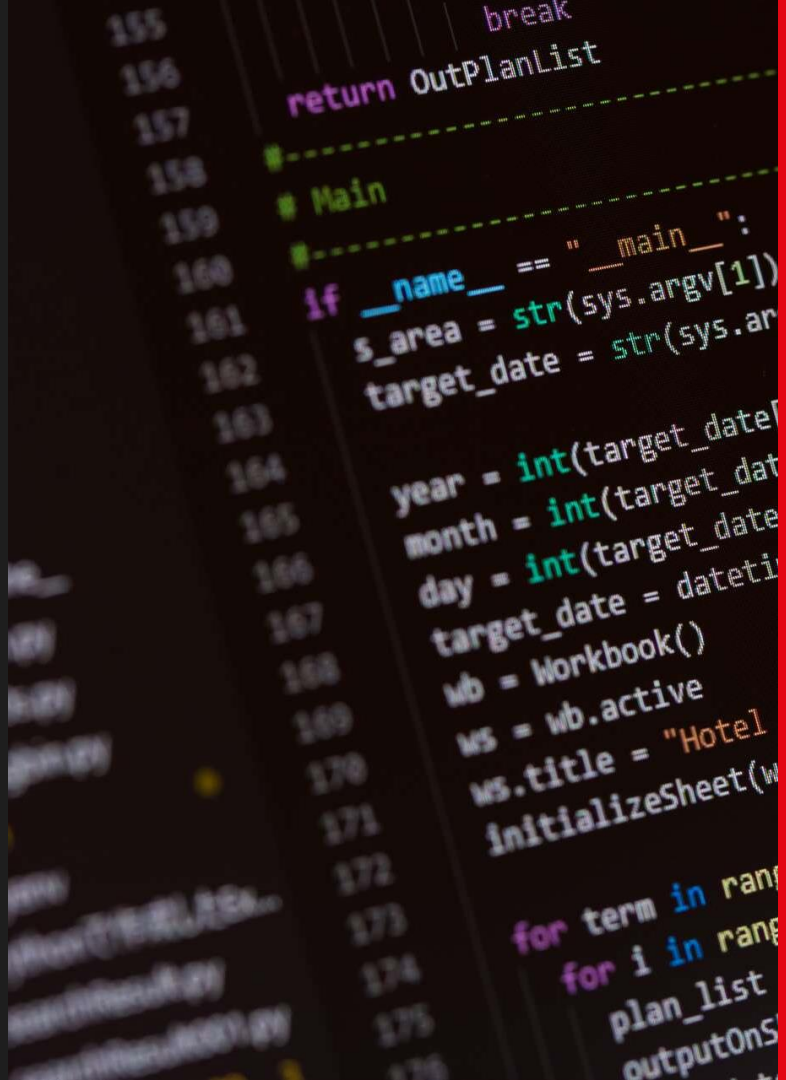
SIG – Systèmes d'Information Géographique

Introduction Python - 1

Gabriel Kathari

Stéphane Joost

GEOME/LGB



Les bases de Python pour le cours de SIG

Les types natifs

Les opérateurs

Les structures conditionnelles

Les boucles

Les types natifs :

- Les nombres (int, float)
- Les booléens (True, False)
- Les chaines de caractères (string)
- Les types séquentiels (listes)

Les nombres (entiers ou flottants)

- Entiers (int) :

```
-5
1
250
135843125498711354154815
```

```
1000000
1_000_000
```

- Flottants (float) :

```
-5.485
1.254
150.87
10.0
```

Les booléens

- **1 : True** ou **0 : False**
- True + 5 = 6 , False + 5 = 5
- Les objets vides ou nul tel que : « », 0, 0.0, [], {} donnent la valeur False

Les chaînes de caractères

```
"Je suis une chaine de caractère"
```

- Délimité par des guillemets simples ou doubles → Préférer des guillemets doubles :

Le guillemet simple reconnu comme une fin de caractère

```
print('on m'appelle l'OVNI')

    print('on m'appelle l'OVNI')
          ^^^^^^^^^^^^
SyntaxError: invalid syntax. Perhaps you forgot a comma?
```

```
print("On m'appelle l'OVNI")
print('On m\'appelle l\'OVNI')
```
→
```
On m'appelle l'OVNI
On m'appelle l'OVNI
```

- Une chaine de caractères peut être multiligne et sera affichée sur plusieurs lignes avec la commande print()

```
string = '''Je suis
une chaine de caractere
sur 3 lignes'''
print(string)
```
→ Output :
```
Je suis
une chaine de caractere
sur 3 lignes
```

- Des caractères spéciaux (\n , \t etc.) → pour ne pas les interpréter il faut ajouter r comme suit :

```
print('C:\Bureau\thierry\nouveau')
```
→
```
C:\Bureau	hierry
ouveau
```

```
print(r'C:\Bureau\thierry\nouveau')
```
→
```
C:\Bureau\thierry\nouveau
```

Les chaines de caractère – fonctions usuelles

- replace()

```
blaze="Eminem"
print(blaze.replace("nem","whinehouse"))
```

→ Emiwhinehouse

- strip() / rstrip() / lstrip()

right left

```
text = "   This is a test string.   "
text = text.strip()
print(text)
```

→ 'This is a test string'

Les espaces ont été enlevés au debut et a la fin.

- split() / join()

```
text = "This is a test string."
words = text.split()
print(words)
```

↓

```
['This', 'is', 'a', 'test', 'string.']
```

```
words = ['This', 'is', 'a', 'test', 'string.']
text = ' '.join(words)
print(text)
```

↓

```
This is a test string.
```

Le séparateur par defaut est l'espace mais n'importe quel caractere peut etre utilisé comme séparateur

Les chaines de caractère – fonctions usuelles

Retournent un booleen

- isdigit() → Tous les caractères sont des chiffres numériques ?
- isupper() → La chaîne de caractère ne contient que des majuscules ?
- istitle() → La chaîne de caractères commence par une majuscule ?
- islower() → La chaînte de caractères ne contient que des minuscules ?

```
text = "12345"
result = text.isdigit()
print(result)
```

True

```
text = "12345abc"
result = text.isdigit()
print(result)
```

False

- stratswith() / endswith()

Ces deux fonctions sont “case sensitive” et différencient les majuscules des minuscules.

```
text = "This is a test string."
result = text.startswith("This")
print(result)
```

True

```
text = "This is a test string."
result = text.startswith("is")
print(result)
```

False

Les constructeurs de types natifs

`str()`	Chaînes de caractères	`str("bonjour")`	`"bonjour"`
`int()`	Nombres entiers	`int(5)`	`5`
`float()`	Nombres décimaux	`float(10.7)`	`10.7`
`bool()`	Booléens	`bool(True)`	`True`

```
str(5)      → "5"
int("2")    → 2
```

```
int("bonjour")  → ValueError: invalid literal for int() with base 10: 'bonjour'
```

La conversion n'est possible que si le changement de type convient a l'objet

Les variables - Nomenclature

- Ne peut pas commencer par un chiffre
- Ne peut pas contenir d'espace
- Ne peut contenir que des caractères alphanumériques (A-z, 0-9) et ne peuvent pas contenir le tiret du haut (-), seulement le tiret du bas (_)
- Certains mots sont réservés (print, True, break etc.)

- Attention : ma_variable ≠ Ma_variable → Sensible a la casse

Les variables - Affectations

- Affectations simples :

```
a = 5
nom = objet
```

- Affectations parallèles :

```
a, b = 5, 8
```

```
a, b, c, d, e, f = 1, 2, 3, 4, 5, 6
```

- Affectations multiples :

```
a = b = c = 5
```

~~`a = b = 5 = c`~~

Les variables – Concaténation 1

- On utilise le « + » pour concaténer deux chaines de caractères.
- On transforme une variable en chaine de caractère avec la fonction str() si la variable le permet :

```
note_intermediaire_SIG = 5.25
note_examen_final_SIG = 5

print("J'ai eu la note de "+str(note_intermediaire_SIG)+" a l'examen intermediaire de SIG")
print("J'ai eu la note de "+str(note_examen_final_SIG)+" a l'examen final de SIG")
```

```
J'ai eu la note de 5.25 a l'examen intermediaire de SIG
J'ai eu la note de 5 a l'examen final de SIG
```

```
note_finale = note_intermediaire_SIG*0.4 + note_examen_final_SIG*0.6

print("Ma note finale est de "+str(note_finale))
```

```
Ma note finale est de 5.1
```

Les operations arithmétiques se font sur les variables de type int et float et non sur les caracterès.

Les variables – Concaténation 2

- f-string:

```
x = 10
y = 12
print(f"la multiplication de {x} par {y} donne {x*y}")
```

```
la multiplication de 10 par 12 donne 120
```

- Méthode format:

```
protocole = "https://"
nom_du_site= "mon_site_web"
extension= "ch"
url = "{}www.{}.{}".format(protocole, nom_du_site,extension)
url = "{}www.{}.{}".format("https://", "mon_site_web","ch")
```

```
https://www.mon_site_web.ch
```

La méthode f-string est intéressante pour un remplissage rapide.
La méthode format() est intéressante pour automatiser un remplissage.

```
note_intermediaire_SIG =  5.25
note_examen_final_SIG = 5

NOTE = "J'ai eu {note_interm} a l'examen intermediaire et {note_finale} a l'examen final"
notification_note = NOTE.format(note_interm=note_intermediaire_SIG, note_finale=note_examen_final_SIG)

print(notification_note)
```

```
J'ai eu 5.25 a l'examen intermediaire et 5 a l'examen final
```

Les opérateurs

- Opérateurs calculatoires basiques (+ * - /)

- + et * permettent aussi d'effectuer des opérations sur les chaines de caractère :

```
print("Eminem"+"50Cent")
    Eminem50Cent
print("Drake"*3)
    DrakeDrakeDrake
```

- Modulo (%) permet de récupérer le reste de la division entière de deux nombres :

```
print(20 % 5)
        0
print(20 % 3)
        2
```

- Division entière (//) : permet de récupérer l'entier de la division.

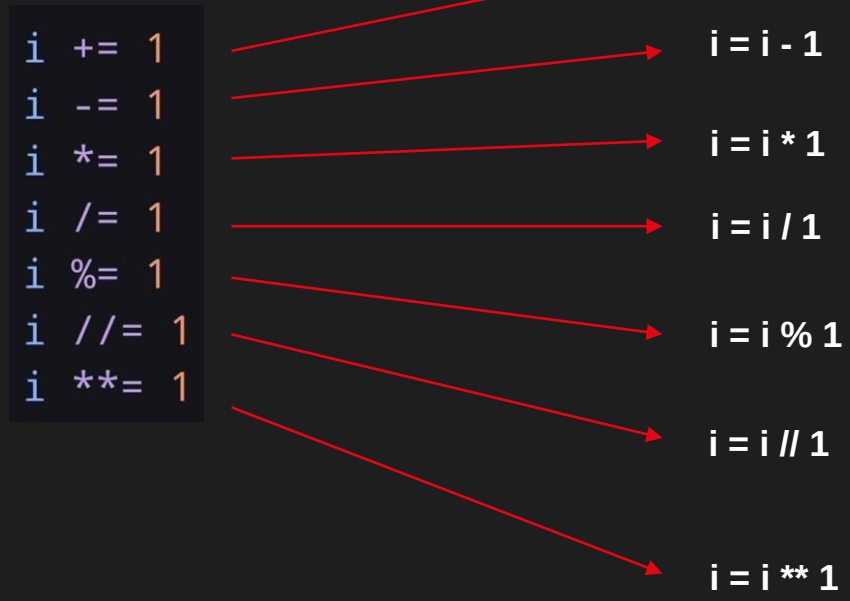
```
print(10 // 3)
        3
```

- L'operateur puissance (**) :

```
print(2 ** 4)
        16
```

Les operateurs - Assignation

- Assignation :



Les operateurs - Comparaison

- Chacun des ses operateurs de comparaison retourne True si la comparaison est respectée et False si ce n'est pas le cas.
- Ces opérateurs de comparaison sont souvent utilisés dans des structures conditionnelles. (cf slide suivante)

- > : plus grand que
- < : plus petit que
- >= : plus grand ou égal a
- <= : plus petit ou égal a
- == : Egal a
- != : Différent de

Exemples :

```
print(2 > 3) ⟶ False
print(3 > 3) ⟶ False
print(3 >= 3) ⟶ True
print(3 <= 3) ⟶ True
print(3 == 3) ⟶ True
print(4 == 3) ⟶ False
print(4 != 3) ⟶ True
print(3 != 3) ⟶ False
```

Les structures conditionnelles

- if / elif /else :

```
if user == "admin":
    print("access allowed")
elif user == "client":
    print("access allowed")
else:
    print("access refused")
```

Outputs :

user = "admin" → access allowed

user = "client" → access allowed

user = "intrus" → access refused

- Opérateurs ternaires : La condition et l'association de la valeur de la variable se font sur la même ligne

```
age = 20
majeur = True if age >= 18 else False
```

→ True

- Operateurs logiques (or and et not) :

```
8 > 3 and (4 < 9 or 10 > 18)
```

True and (True or False)

True and True

True

```
name = "Luca"
if not name=="Jhon":
    print("Le nom est incorrect")
```

Output :

Le nom est incorrect

Les listes

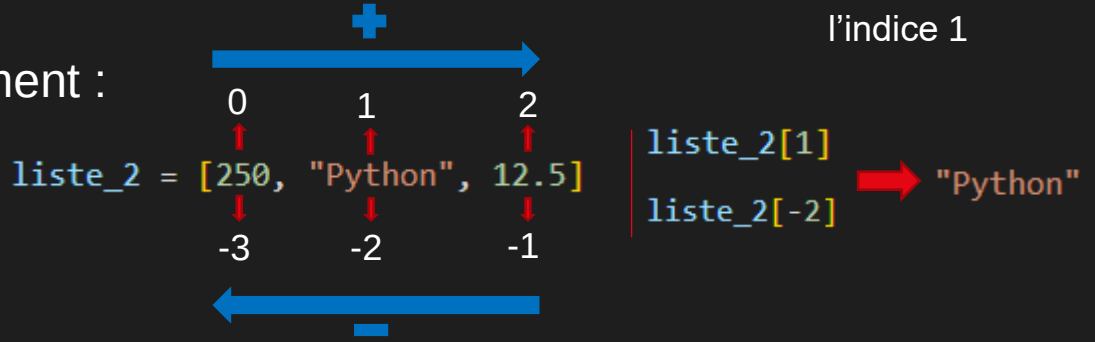
- Une liste peut être composée d'éléments de type différent :

```
liste_1 = [1,2,3,4,5,6]
```

```
liste_2 = [250, "Python", 12.5]
```

L'indice correspond a la position de l'élément dans la liste de 0 a n

- Ajouter un élément : `liste_1.append(7)` `liste_2.append("C++")`
- Ajouter une liste : `liste_2.extend(["Java","React","Angular"])`
- Retirer un élément (pas l'indice) : `liste_2.remove("Python")`
- Retirer un élément par son indice : `liste_2.pop(1)` → Enlève l'élément a l'indice 1
- Accéder a un élément :



Les listes

- Accéder a une slice de la liste :

```
liste_3 = ["A2H","Kaaris","B2O","Heuss","Ninho","JUL"]
```

```
liste_3[1:3]  ➡  "Kaaris","B2O"
```

On commence avec l' élément d'index 1 et on s'arrête avant l'index 3

```
liste_3[:]  ➡  "A2H","Kaaris","B2O","Heuss","Ninho","JUL"
```

On prend tous les éléments de la liste

```
liste_3[:-2]  ➡  "A2H","Kaaris","B2O","Heuss"
```

On prend tous les éléments depuis le debut et on s'arrête avant l'indice -2

```
liste_3[0:5:2]  ➡  "A2H","B2O","Ninho"
```

On commence à l'indice 0

On fini avant l'indice 5

On fait un pas de 2 donc on prend un élément sur 2

Les listes

- Opérateurs d'appartenance (in / not in):

```
users = ["Julie", "Annie","Marc"]

if "Paul" in users:
    print("C'est un utilisateur")

if "Paul" not in users:
    print("Ce n'est pas un utilisateur")
```

Output :

```
Ce n'est pas un utilisateur
```

- Liste imbriquée :

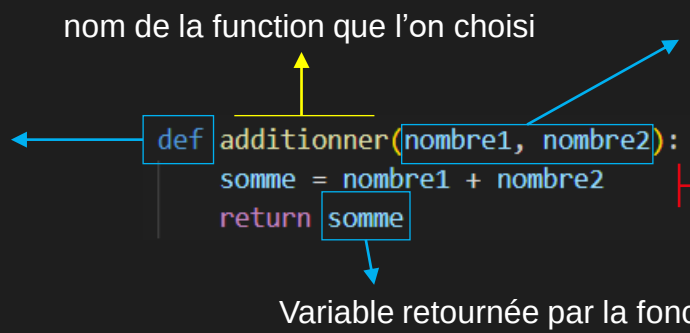
```
liste = ["Pierre", ["Marie","Julie",["Aude"]],"Emric"]

liste[0]    → "Pierre"
liste[1][1] → "Julie"
liste[1][2] → ["Aude"]
liste[1][2][0] → "Aude"
```

Méthode vs fonction

```
def additionner(nombre1, nombre2):
    somme = nombre1 + nombre2
    return somme
```

- Déclaration d'une fonction :



- Appel d'une fonction :

Arguments donnés à la function : nombre1 ← 5, nombre2 ← 3

Resultat de la function
resultat ← somme

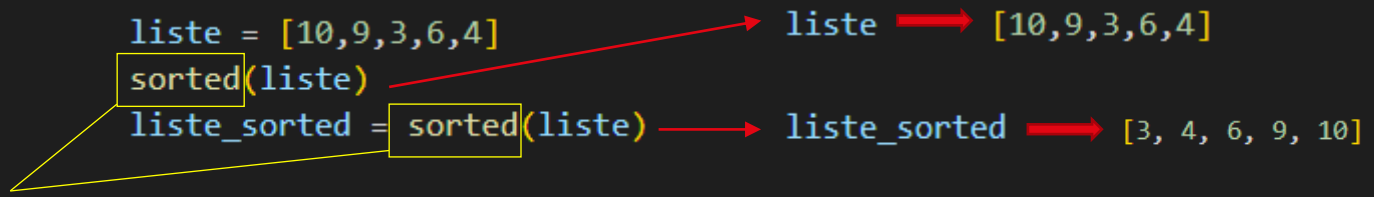
```
resultat = additionner(5, 3)

print("Le résultat de l'addition est :", resultat)
```

Méthode vs fonction

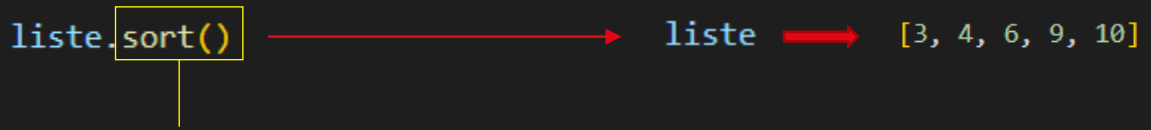
- Une méthode est une fonction qui appartient a un objet.

Fonction :



Pour une fonction, il faut écraser ou assigner a une autre variable la liste sur laquelle elle s'applique pour sauvegarder la modification

Methode :



Une methode modifie directement l'objet sur lequel elle est appliquée

Ce n'est pas valable pour les chaines de caracteres ou les nombres (objets immuables)

Méthodes et fonctions utiles

- len()

```
len("Mathilde") → 8
len([1,2,3,4]) → 4
```

- round()

```
round(10.2) → 10
round(10.7) → 11
```

- min() / max()

```
min([1,2,3,4,5]) → 1
max([1,2,3,4,5]) → 5
```

```
min("abcde") → "a"
max("abcde") → "e"
```

- sum()

```
sum([1,2,3]) → 6
sum([1,2,"jul"]) → error
```

Ne fonctionne que sur des nombres

- range()

```
range(5) → [0,1,2,3,4]
range(2,5) → [2,3,4]
```

Les boucles - for

- Sur une liste :

```
for element in liste:
    #effectuer des operations avec element
    print("cacluls sur element finis")
```

5 iterations

```
for i in [0,1,3,5,8]:
    print(i)
```

Output :

```
0
1
3
5
8
```

- Sur une chaine de caractères :

```
for lettre in mot:
    print(lettre)
```

6 iterations

```
for lettre in "Python":
    print(lettre)
```

Output :

```
P
y
t
h
o
n
```

- Répéter x (1000 ici) fois une opération :

```
for i in range(1000):
    #Effectuer l'operation
    print(i)
```

Les boucles - for

- La fonction enumerate() permet de récupérer le compte de chaque itération de al boucle de 0 a n-1 éléments de la liste comme suit :

```
liste = ["rouge","blanc","vert","orange"]

for count, color in enumerate(liste):
    print(str(count)+" "+color)
```

Output :

```
0 rouge
1 blanc
2 vert
3 orange
```

- La variable count commence a 0 et est incrémentée de 1 a chaque nouvelle itération de la boucle. On peut aussi utiliser cette variable comme argument pour sortir de la boucle.

```
for count, color in enumerate(liste):
    print(str(count)+" "+color)
    if count >=2:
        break
```

Output :

```
0 rouge
1 blanc
2 vert
```

Les boucles - while

- Répéter x (1000 ici) fois une opération :

```
i=0
while i < 1000:
    print(i)
    i += 1
```

A chaque iteration l'output sera la valeur de l'indice tel que :

Output :

```
0
1
2
...
997
998
999
```

- Arrêter la boucle si la condition n'est plus remplie :

```
go_to_next = "y"
while go_to_next == "y":
    # Do calculation
    go_to_next = input("Do you want to go to next iteration ? y/n")
    print("We go to next iteration")
```

A chaque iteration l'output sera : `we go to next iteration`

Les boucles – continue et break

- **continue**: cette instruction permet de passer directement a la prochaine itération. Elle est souvent associée a une condition dans la boucle.

Si element est un nombre on passe à l' itération suivante

```
liste = ["1","2","a","3","b"]
for element in liste:
    if element.isdigit():
        continue
    print(element)
```

Output :

```
a
b
```

- **break** : cette instruction permet d'immédiatement quitter la boucle.

On quitte la boucle seulement quand element n'est pas un nombre

```
liste = ["1","2","a","3","b"]
for element in liste:
    if not element.isdigit():
        break
    print(element)
```

Output :

```
1
2
```

Les boucles – listes en compréhension

- Ces listes permettent d'écrire de façon compacte des actions simples itérées sur une liste :

```
liste = ["1","2","a","3","b","c"]
chiffres = [i for i in liste if i.isdigit()]
print(chiffres)
```

Output :

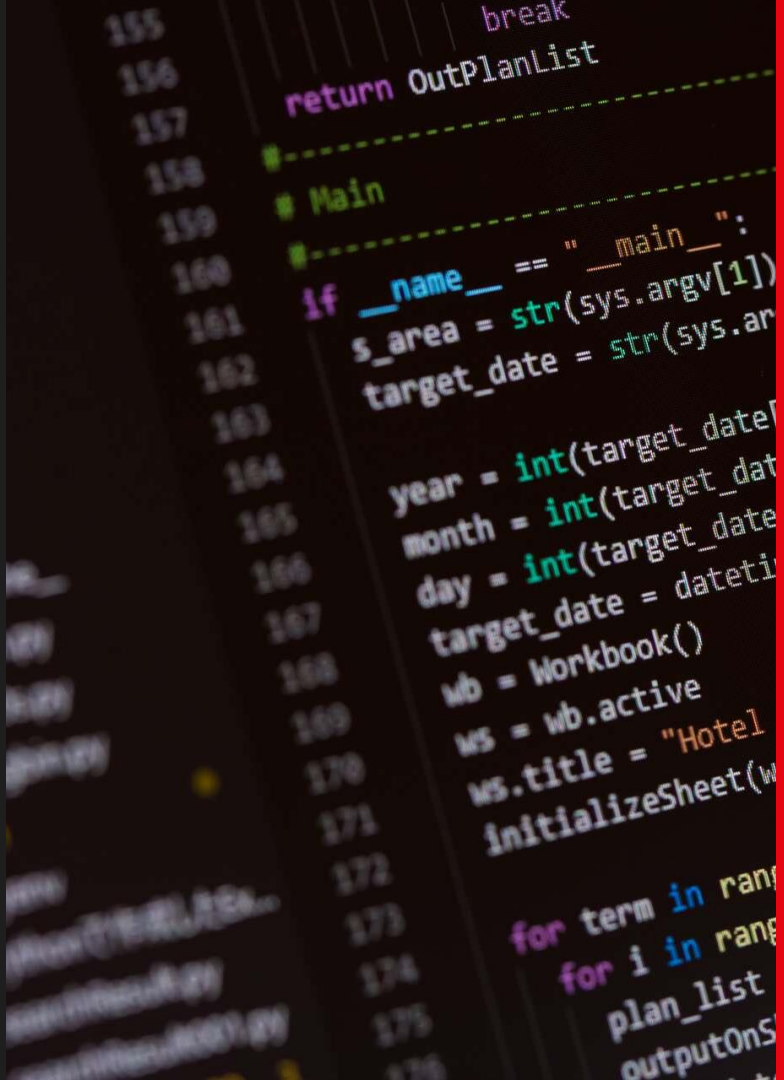
```
['1', '2', '3']
```

- Il est aussi possible de modifier directement la valeur de l' élément dans la liste en compréhension :

```
liste = [-2,-1,0,1,2,3,3,4,5]
chiffres_positifs = [i for i in liste if i>0]
chiffres_positifs = [i*2 for i in liste if i>0]
```

Output :

```
[1, 2, 3, 3, 4, 5]
[2, 4, 6, 6, 8, 10]
```



Merci pour votre attention !