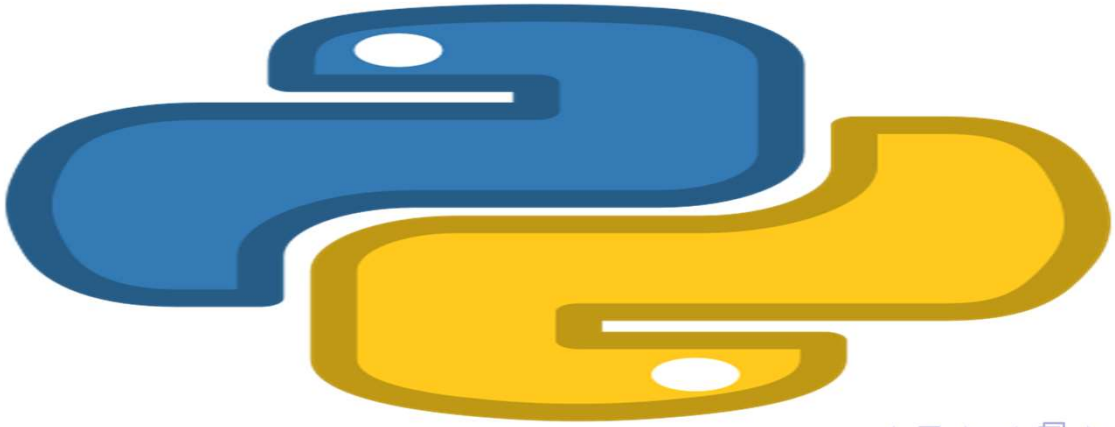
Python : modules et packages

Mohammed OUANAN

m.ouanan@umi.ac.ma



Plan

Python

Dans une application **Python**

- On peut utiliser des éléments définis dans un autre fichier : une variable, une fonction, une classe, une interface...
- Pour cela, il faut l'importer là où on a besoin de l'utiliser
- Un fichier contenant du code **Python** = `module`

Quatre types de modules

- **Built-in modules** :
  - Définis dans la librairie standard de **Python** : **STDLIB**.
  - Pour les utiliser, ils ne nécessitent aucune importation.
- **Core modules** :
  - Définis dans **STDLIB**.
  - Pour les utiliser, il faut les importer.
- **Community modules** :
  - Proposés par la communauté **Python**.
  - Pour les utiliser, il faut les installer et ensuite les importer.
- **Custom modules** :
  - Définis par le développeur.
  - Pour les utiliser, il faut les importer.

Python

Avant de commencer

- Créez un nouveau projet `python_modules` dans votre espace de travail
- Créez un fichier `main.py`
- Validez

Python

Étant donné le fichier `fonctions.py` ayant le contenu suivant

```
somme = lambda a, b: a + b

produit = lambda a, b: a * b
```

Python

Étant donné le fichier `fonctions.py` ayant le contenu suivant

```
somme = lambda a, b: a + b

produit = lambda a, b: a * b
```

Pour importer le contenu de `fonctions.py` dans `main.py`

```
import fonctions

print(fonctions.somme (2, 3))
# affiche 5

print(fonctions.produit (2, 3))
# affiche 6
```

Python

On peut aussi utiliser des alias

```
import fonctions as f

print(f.somme (2, 3))
# affiche 5

print(f.produit (2, 3))
# affiche 6
```

Python

On peut aussi indiquer ce que l'on souhaite importer et simplifier l'utilisation

```
from fonctions import somme, produit

print(somme (2, 3))
# affiche 5

print(produit (2, 3))
# affiche 6
```

Python

On peut aussi utiliser les alias pour les fonctions

```
from fonctions import somme as s, produit as p

print(s (2, 3))
# affiche 5

print(p (2, 3))
# affiche 6
```

Python

Pour tout importer, on peut utiliser *

```
from fonctions import *

print(somme (2, 3))
# affiche 5

print(produit (2, 3))
# affiche 6
```

Python

Packages

- Un répertoire contenant des fichiers et/ou répertoires est un package.
- Avant **Python 3.3**, un package contenant des modules **Python** devait contenir un fichier `__init__.py`, indiquant à **Python** qu'il s'agissait d'un package.
- Ce fichier est désormais facultatif, mais reste utile pour initialiser des variables de package.

Python

Packages

- Un répertoire contenant des fichiers et/ou répertoires est un package.
- Avant **Python 3.3**, un package contenant des modules **Python** devait contenir un fichier `__init__.py`, indiquant à **Python** qu'il s'agissait d'un package.
- Ce fichier est désormais facultatif, mais reste utile pour initialiser des variables de package.

Pour la suite

- créons un répertoire appelé `package` à la racine du projet.
- déplaçons le fichier `fonctions.py` dans `package`.

Python

Dans `main.py`, pour importer et utiliser les fonctions définies dans `fonctions.py`

```
from package.fonctions import produit, somme

print(somme (2, 3))
# affiche 5

print(produit (2, 3))
# affiche 6
```

Python

Et si `fonctions.py` **était dans** `subpackage` **qui est défini dans** `package`

```
from package.subpackage.fonctions import produit, somme

print(somme (2, 3))
# affiche 5

print(produit (2, 3))
# affiche 6
```

Python

Constantes pour modules et packages

- `__name__` contient
  - soit la valeur `__main__` dans le fichier d'entrée
  - soit le chemin complet depuis la racine du projet vers le fichier source
- `__package__` contient le package et les sous-packages du fichier source
- `__file__` contient le chemin complet depuis la racine du disque vers le fichier source

Python

En plaçant les constantes précédentes dans `main.py`, le résultat est

```
print(__name__)
# affiche __main__

print(__package__)
# affiche None

print(__file__)
# affiche C:/Users/user/cours-modules/main.py
```

Python

En les plaçant dans `fonctions.py`, le résultat est

```
print(__name__)
# affiche package.subpackage.fonctions

print(__package__)
# affiche package.subpackage

print(__file__)
# affiche C:\Users\user\cours-modules\package\subpackage\fonctions.py
```

Python

Dans `fonctions.py`, ajoutons la fonction `somme_carre`

```
somme = lambda a, b: a + b

produit = lambda a, b: a * b

somme_carre = lambda a, b: somme(a * a, b * b)

print(somme_carre(3, 4))
```

Python

Dans `fonctions.py`, ajoutons la fonction `somme_carre`

```
somme = lambda a, b: a + b

produit = lambda a, b: a * b

somme_carre = lambda a, b: somme(a * a, b * b)

print(somme_carre(3, 4))
```

En exécutant `fonctions.py`, le résultat est :

```
25
```

Python

En exécutant `main.py`

```
from package.subpackage.fonctions import *

print(somme (2, 3))

print(produit (2, 3))
```

Python

En exécutant `main.py`

```
from package.subpackage.fonctions import *

print(somme (2, 3))

print(produit (2, 3))
```

Le résultat est :

```
25
5
6
```

Python

Modifions `fonctions.py` **pour ne plus exécuter** `somme_carre` **que si on exécute** `fonctions.py`

```
somme = lambda a, b: a + b

produit = lambda a, b: a * b

somme_carre = lambda a, b: somme(a * a, b * b)

if __name__ == '__main__':
    print(somme_carre(3, 4))
```

Python

En exécutant `fonctions.py`, le résultat est :

```
25
```

Python

En exécutant `fonctions.py`, le résultat est :

```
25
```

En exécutant `main.py`, le résultat est :

```
5
6
```

Python

Import relatif en **Python**

- Commence par `.` (répertoire actuel) ou `..` (répertoire parent) pour naviguer dans la structure du package.
- Autorisé uniquement si le module n'est pas exécuté comme script principal : ayant un `__name__` différent de `__main__`

Python

Dans `package/subpackage`**, considérons le module** `avancees.py`

```
def somme_carree(a, b):
    return somme(a*a, b*b)
```

Python

Dans `package/subpackage`, considérons le module `avancees.py`

```
def somme_carree(a, b):
    return somme(a*a, b*b)
```

La fonction `somme` peut être importer avec un chemin absolu

```
from package.subpackage.fonctions import somme


def somme_carree(a, b):
    return somme(a*a, b*b)
```

Python

Dans `package/subpackage`**, considérons le module** `avancees.py`

```
def somme_carree(a, b):
    return somme(a*a, b*b)
```

La fonction `somme` **peut être importer avec un chemin absolu**

```
from package.subpackage.fonctions import somme


def somme_carree(a, b):
    return somme(a*a, b*b)
```

Pour tester dans `main.py`

```
from package.subpackage.avancees import somme_carree

print(somme_carree(2, 3))
# affiche 13
```

Python

La fonction `somme` peut être importer avec un chemin relatif

```
from .fonctions import somme


def somme_carree(a, b):
    return somme(a*a, b*b)
```

Python

La fonction `somme` peut être importer avec un chemin relatif

```
from .fonctions import somme


def somme_carree(a, b):
    return somme(a*a, b*b)
```

`main.py` **reste inchangé**

```
from package.subpackage.avancees import somme_carree

print(somme_carree(2, 3))
# affiche 13
```

Python

Deuxième import relatif possible

```
from . import fonctions


def somme_carree(a, b):
    return fonctions.somme(a*a, b*b)
```

Python

Deuxième import relatif possible

```
from . import fonctions


def somme_carree(a, b):
    return fonctions.somme(a*a, b*b)
```

`main.py` reste inchangé

```
from package.subpackage.avancees import somme_carree

print(somme_carree(2, 3))
# affiche 13
```

Python

Remarque

Déplaçons `fonctions.py` dans un package `src/utils`.

Python

Pour utiliser `somme` avec un import relatif

```
from ..utils import fonctions


def somme_carree(a, b):
    return fonctions.somme(a*a, b*b)
```

Python

Pour utiliser `somme` avec un import relatif

```
from ..utils import fonctions


def somme_carree(a, b):
    return fonctions.somme(a*a, b*b)
```

`main.py` **reste inchangé**

```
from package.subpackage.avancees import somme_carree

print(somme_carree(2, 3))
# affiche 13
```

Python

Dans `package/subpackage/__init__.py`**, ajoutons l'**`import` **suivant**

```
from .avancees import somme_carree
```

Python

Dans `package/subpackage/__init__.py`**, ajoutons l'**`import` **suivant**

```
from .avancees import somme_carree
```

Ainsi, l'`import` **dans** `main.py` **peut être simplifié**

```
from package.subpackage import somme_carree

print(somme_carree(2, 3))
# affiche 13
```

Python

Modifions `package/subpackage/__init__.py` **en ajoutant** `__all__`

```
from .avancees import somme_carree

__all__ = ["somme_carree"]
```

Python

Modifions `package/subpackage/__init__.py` **en ajoutant** `__all__`

```
from .avancees import somme_carree

__all__ = ["somme_carree"]
```

Ainsi, nous pouvons utiliser l'`import *`

```
from package.subpackage import *

print(somme_carree(2, 3))
# affiche 13
```

Python

Dans `package/__init__.py`**, ajoutons le code suivant**

```
from .subpackage import somme_carree

__all__ = ["somme_carree"]
```

Python

Dans `package/__init__.py`**, ajoutons le code suivant**

```
from .subpackage import somme_carree

__all__ = ["somme_carree"]
```

Ainsi, l'`import` **dans** `main.py` **peut être simplifié encore plus**

```
from package import *

print(somme_carree(2, 3))
# affiche 13
```

Python

Bonnes pratiques d'organisation d'un projet **Python**

- Un seul module d'entrée à la racine du projet, souvent nommé `main.py`, pour centraliser le lancement du projet.
- Un package, par exemple `src`, à la racine du projet pour contenir le code source, organisé en sous-packages.
- Un fichier `requirements.txt` pour les dépendances du projet.
- Un fichier `README.md` pour documenter le projet et ses instructions d'utilisation.
- Un répertoire `tests` pour regrouper les tests unitaires.

STDLIB

Python

STDLIB

- Librairie standard de **Python**
- Collection de modules et de packages intégrés : disponibles dès l'installation de **Python**
- Conçu pour simplifier le développement de logiciels
- Offrant des outils pour des tâches courantes

Python

Quelques modules **STDLIB** : fonctions mathématiques

- `math` pour les fonctions arithmétiques
- `random` pour la génération de nombres aléatoires
- `decimal` pour une meilleure manipulation de nombres décimaux
- `statistics` pour les fonctions statistiques de base
- ...

Python

Quelques modules **STDLIB** : manipulation de données

- `datetime` : Manipulation des dates et des heures.
- `json` : Encodage et décodage de données **JSON**.
- `csv` : Lecture et écriture de fichiers au format **CSV**.
- ...

Python

Quelques modules **STDLIB** : fonctionnalités de Fichiers et d'**OS**

- `glob` pour la recherche de fichiers/répertoires
- `os` pour réaliser des opérations sur le système d'exploitation
- `os.path` pour manipuler les chemins de fichiers
- `pathlib` pour la manipulation orientée objet des chemins de fichiers
- `sys` pour effectuer des opérations d'entrée et sortie plus personnalisées
- `shutil` pour réaliser des opérations sur les fichiers
- ...

Python

Quelques modules **STDLIB** : réseau et communication

- `socket` : Interface de bas niveau pour les communications réseau
- `http` : Modules pour travailler avec **HTTP**
  - `http.client` pour le client
  - `http.server` pour le serveur
- `urllib` : Collection de modules pour travailler avec les URLs.
- ...

Python

Quelques modules **STDLIB** : **Q**ualité des **A**pplications (**QA**)

- `unittest`
- `doctest`

Python

Quelques modules **STDLIB** : **Q**ualité des **A**pplications (**QA**)

- `unittest`
- `doctest`

Quelques modules **STDLIB** : développement d'interface graphique (**GUI**)

tkinter : développement d'applications desktop.

Python

Fonctions mathématiques prédéfinies (**Built-in functions**)

- `abs(x)` : retourne la valeur absolue de `x`
- `pow(x, y)` : retourne `x` puissance `y`
- `max(x, y)` : retourne le max de `x` et `y`
- `min(x, y)` : retourne le min de `x` et `y`
- `round(x)` : retourne l'arrondi de `x`

Python

Fonctions mathématiques prédéfinies (**Built-in functions**)

- `abs(x)` : retourne la valeur absolue de `x`
- `pow(x, y)` : retourne `x` puissance `y`
- `max(x, y)` : retourne le max de `x` et `y`
- `min(x, y)` : retourne le min de `x` et `y`
- `round(x)` : retourne l'arrondi de `x`

Exemple

```
print(max(1, abs(-3), 2))
# affiche 3
```

Python

Fonctions mathématiques nécessitant l'importation du module `math`

- `sqrt(x)` : retourne la racine carré de `x`
- `trunc(x)` : retourne la partie entière de `x`
- `fsum(iterable)` : retourne la somme de tous les éléments de `iterable`
- `factorial(x)` : retourne la factorielle de `x`
- `floor(x)` et `ceil(x)` : retournent l'arrondi de `x`
- ...

Python

Fonctions mathématiques nécessitant l'importation du module `math`

- `sqrt(x)` : retourne la racine carré de `x`
- `trunc(x)` : retourne la partie entière de `x`
- `fsum(iterable)` : retourne la somme de tous les éléments de `iterable`
- `factorial(x)` : retourne la factorielle de `x`
- `floor(x)` et `ceil(x)` : retournent l'arrondi de `x`
- ...

Exemple

```
import math

print(math.sqrt(25))
# affiche 5.0
```

Python

Exemple avec `math.floor(x)`, `math.ceil(x)` **et** `round(x)`

```
print(round(1.9))
# affiche 2
print(round(1.5))
# affiche 2
print(round(1.4))
# affiche 1
print(math.ceil(1.9))
# affiche 2
print(math.ceil(1.5))
# affiche 2
print(math.ceil(1.4))
# affiche 2
print(math.floor(1.9))
# affiche 1
print(math.floor(1.5))
# affiche 1
print(math.floor(1.4))
# affiche 1
```

Python

Pour extraire la partie entière et la partie décimale, on utilise la fonction `modf` (modulus and fraction)

```
frac, ent = math.modf(2.5)

print(frac)
# affiche 0.5

print(ent)
# affiche 2.0
```

Exercice

Écrire une fonction `equation_second_degre(a, b, c)` qui permet de résoudre une équation de second degré ($ax^2 + bx + c = 0$)

- $\Delta = b^2 - 4ac$
- Si $\Delta < 0$: pas de solution
- Si $\Delta = 0$: une solution $-b/2a$
- Si $\Delta > 0$: deux solutions $(-b - \sqrt{\Delta})/2a$ et $(-b + \sqrt{\Delta})/2a$

Résultat attendu

```
print(equation_second_degre(1, 1, -2))
# affiche (-2.0, 1.0)

print(equation_second_degre(1, 2, 1))
# affiche -1.0

print(equation_second_degre(2, 3, 5))
# affiche pas de solution
```

Python

Solution

```
import math

delta = lambda a, b, c : pow(b, 2) - 4 * a * c

def equation_second_degre(a, b, c):
    d = delta(a, b, c)
    if d < 0:
        return "pas de solution"
    elif d == 0:
        return -b / (2 * a)
    else:
        return (-b - math.sqrt(d)) / (2 * a), (-b + math.sqrt(d)) / (2 * a)

print(equation_second_degre(1, 1, -2))
# affiche (-2.0, 1.0)

print(equation_second_degre(1, 2, 1))
# affiche -1.0

print(equation_second_degre(2, 3, 5))
# affiche pas de solution
```

Python

Fonctions mathématiques nécessitant l'importation du module `random`

- `random()` : retourne un nombre réel aléatoire entre 0 et 1
- `randint(x, y)` : retourne un nombre entier aléatoire entre `x` et `y` (`x` et `y` inclus)
- `choice(iterable)` : retourne un élément aléatoire appartenant à `iterable`
- ...

Python

Fonctions mathématiques nécessitant l'importation du module `random`

- `random()` : retourne un nombre réel aléatoire entre 0 et 1
- `randint(x, y)` : retourne un nombre entier aléatoire entre `x` et `y` (`x` et `y` inclus)
- `choice(iterable)` : retourne un élément aléatoire appartenant à `iterable`
- ...

Exemple

```
import random

print(random.randint(5, 10))
# affiche un nombre entre 5 et 10
```

Python

Le module `operator` fournit un ensemble de fonctions

- qui correspondent aux opérations intrinsèques du langage, telles que les opérations arithmétiques, les comparaisons et les opérations sur les séquences.
- souvent utilisées comme callback par d'autres modules comme `functools`.

Python

Opérations arithmétiques fournies par `operator`

- `operator.add(a, b)` : Additionne `a` et `b` (équivalent à `a + b`).
- `operator.sub(a, b)` : Soustrait `b` de `a` (équivalent à `a - b`).
- `operator.mul(a, b)` : Multiplie `a` et `b` (équivalent à `a * b`).
- `operator.truediv(a, b)` : Divise `a` par `b` (équivalent à `a / b`).

Python

Opérations arithmétiques fournies par `operator`

- `operator.add(a, b)` : Additionne `a` et `b` (équivalent à `a + b`).
- `operator.sub(a, b)` : Soustrait `b` de `a` (équivalent à `a - b`).
- `operator.mul(a, b)` : Multiplie `a` et `b` (équivalent à `a * b`).
- `operator.truediv(a, b)` : Divise `a` par `b` (équivalent à `a / b`).

Exemple

```
import operator

print(operator.add(2, 5))
# affiche 7
```

Python

Opérations de comparaison fournies par `operator`

- `operator.eq(a, b)` : Vérifie si `a` est égal à `b` (équivalent à `a == b`).
- `operator.ne(a, b)` : Vérifie si `a` est différent de `b` (équivalent à `a != b`).
- `operator.lt(a, b)` : Vérifie si `a` est inférieur à `b` (équivalent à `a < b`).
- `operator.le(a, b)` : Vérifie si `a` est inférieur ou égal à `b` (équivalent à `a <= b`).
- `operator.gt(a, b)` : Vérifie si `a` est supérieur à `b` (équivalent à `a > b`).
- `operator.ge(a, b)` : Vérifie si `a` est supérieur ou égal à `b` (équivalent à `a >= b`).

Python

Opérations de comparaison fournies par `operator`

- `operator.eq(a, b)` : Vérifie si `a` est égal à `b` (équivalent à `a == b`).
- `operator.ne(a, b)` : Vérifie si `a` est diffèrent de `b` (équivalent à `a != b`).
- `operator.lt(a, b)` : Vérifie si `a` est inférieur à `b` (équivalent à `a < b`).
- `operator.le(a, b)` : Vérifie si `a` est inférieur ou égal à `b` (équivalent à `a <= b`).
- `operator.gt(a, b)` : Vérifie si `a` est supérieur à `b` (équivalent à `a > b`).
- `operator.ge(a, b)` : Vérifie si `a` est supérieur ou égal à `b` (équivalent à `a >= b`).

Exemple

```
import operator

print(operator.gt(2, 5))
# affiche False
```

Python

string

Utilisé pour fournir des constantes et des fonctions utiles pour manipuler des chaînes de caractères

- **Constantes utiles** : comme `string.ascii_letters`, `string.digits`, `string.punctuation`...
- **Méthodes de formatage** : fournit des fonctions pour formater des chaînes de caractères, comme `string.format()`, `string.Template`...

Python

Quelques problèmes avec `float`

```
import string
```

Python

Quelques problèmes avec `float`

```
import string
```

Exemple avec les constantes

```
print(string.ascii_lowercase)
# affiche abcdefghijklmnopqrstuvwxyz

print(string.digits)
# affiche 0123456789

print(string.ascii_letters)
# affiche abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Python

decimal

- Module **Python** permettant de manipuler des nombres à virgules dont le calcul est plus exacte en comparaison avec le type `float`
- Type conçu principalement pour les utilisateurs et pas pour la machine

Python

decimal

- Module **Python** permettant de manipuler des nombres à virgules dont le calcul est plus exacte en comparaison avec le type `float`
- Type conçu principalement pour les utilisateurs et pas pour la machine

Quelques problèmes avec `float`

```
print(0.1 + 0.1 + 0.1 - 0.3)
# affiche 5.551115123125783e-17

print(1.1 + 2.2)
# affiche 3.3000000000000003
```

Python

On peut construire un décimal à partir d'un entier ou une chaîne de caractères

```
from decimal import *

print(Decimal(3))
# affiche 3

print(Decimal("3.33"))
# affiche 3.33

print(Decimal(str(2.0 ** 2)))
# affiche 4.0
```

Python

On peut construire un décimal à partir d'un entier ou une chaîne de caractères

```
from decimal import *

print(Decimal(3))
# affiche 3

print(Decimal("3.33"))
# affiche 3.33

print(Decimal(str(2.0 ** 2)))
# affiche 4.0
```

On peut aussi construire un décimal à partir d'un float (mais il est conseillé de passer par les chaînes)

```
print (Decimal(3.3))
# affiche 3.2999999999999998223643160599749535322189331054687 5

print (Decimal(str(3.3)))
# affiche 3.3
```

Python

On peut modifier la précision

```
x = Decimal('2.4444')
y = Decimal('3.4321')

print(x + y)
# affiche 5.8765

getcontext().prec = 3
print(x + y)
# affiche 5.88

getcontext().prec = 2
print(x + y)
# affiche 5.9

getcontext().prec = 1
print(x + y)
# affiche 6
```

Python

fraction

- Module **Python** permettant de manipuler des nombres rationnels
- Construction possible depuis une paire d'entiers, un autre nombre rationnel, ou une chaîne de caractères.

Python

Exemples de construction de fraction

```
from decimal import Decimal
from fractions import Fraction

print(Fraction(5, 2))
# affiche 5/2

print(Fraction(2.5))
# affiche 5/2

print(Fraction(Decimal("2.5")))
# affiche 5/2

print(Fraction(5 / 2))
# affiche 5/2

print(Fraction("5/2"))
# affiche 5/2
```

Python

Pour extraire le numérateur ou le dénominateur

```
x = Fraction(2.5)

print(x.numerator)
# affiche 5

print(x.denominator)
# affiche 2
```

Les opérateurs arithmétiques peuvent être appliqués sur les fractions

```
f1 = Fraction(1, 2)
f2 = Fraction(3, 4)

# Addition
result_add = f1 + f2
print(result_add)
# affiche 5/4

# Soustraction
result_sub = f1 - f2
print(result_sub)
# affiche -1/4

# Multiplication
result_mul = f1 * f2
print(result_mul)
# affiche 3/8

# Division
result_div = f1 / f2
print(result_div)
# affiche 2/3
```

Python

Tableaux statiques : `array`

- Structure de données acceptant plusieurs valeurs de même type
- Type à spécifier à la création du tableau en utilisant le type code
- Seuls les types primitifs suivants sont acceptés : caractères, entiers et flottants

Python

Quelques types codes

- `b` : pour les caractères signés (byte : nombre compris entre -128 et 127)
- `B` : pour les caractères non-signés (byte : combre compris entre 0 et 255)
- `h` : pour les shorts signés
- `H` : pour les shorts non-signés
- `i` : pour les entiers signe´s
- `I` : pour les entiers non-signe´s
- `l` : pour les longs signe´s
- `L` : pour les longs non-signe´s
- `f` : pour les flottants
- `d` : pour les doubles

Python

Pour utiliser les tableaux statiques, il faut faire l'import suivant

```
from array import array
```

Python

Pour utiliser les tableaux statiques, il faut faire l'import suivant

```
from array import array
```

Pour déclarer un tableau

```
arr = array('i')
```

Python

Pour utiliser les tableaux statiques, il faut faire l'import suivant

```
from array import array
```

Pour déclarer un tableau

```
arr = array('i')
```

Pour déclarer un tableau avec quelques valeurs initiales

```
arr = array('i', [2, 3, 8, -5])
```

Python

Pour utiliser les tableaux statiques, il faut faire l'import suivant

```
from array import array
```

Pour déclarer un tableau

```
arr = array('i')
```

Pour déclarer un tableau avec quelques valeurs initiales

```
arr = array('i', [2, 3, 8, -5])
```

Pour afficher les caractéristiques du tableau

```
print(arr)
# affiche array('i', [2, 3, 8, -5])
```

Python

Pour connaître le nombre d'éléments d'un tableau

```
print(len(arr))
# affiche 4
```

Python

Pour connaitre le nombre d'éléments d'un tableau

```
print(len(arr))
# affiche 4
```

Pour ajouter un élément (à la fin)

```
arr.append(10)

print(arr)
# affiche array('i', [2, 3, 8, -5, 10])
```

Python

Pour connaître le nombre d'éléments d'un tableau

```
print(len(arr))
# affiche 4
```

Pour ajouter un élément (à la fin)

```
arr.append(10)

print(arr)
# affiche array('i', [2, 3, 8, -5, 10])
```

Pour ajouter un élément à une position donnée (ici 2)

```
arr.insert(2, 6)

print(arr)
# affiche array('i', [2, 3, 6, 8, -5])
```

Python

Pour récupérer l'indice de la première occurrence d'un élément dans le tableau, on utilise `index`. Si l'élément n'existe pas, la méthode lève une exception.

```
print(arr.index(8))
# affiche 2
```

Python

Pour récupérer l'indice de la première occurrence d'un élément dans le tableau, on utilise `index`. Si l'élément n'existe pas, la méthode lève une exception.

```
print(arr.index(8))
# affiche 2
```

Pour supprimer un élément, on utilise `remove`. Si l'élément n'existe pas, la méthode lève une exception.

```
arr.remove(3)

print(arr)
# affiche array('i', [2, 8, -5])
```

Python

Autres opérations sur les tableaux statiques

- `count(x)` renvoi le nombre d'occurrences de `x` dans le tableau.
- `fromlist(list)` ajoute les éléments de la liste à notre tableau
- `pop([i])` supprime l'élément d'indice `i` du tableau
- `reverse()` inverse l'ordre des éléments du tableau.
- `tolist()` convertit le tableau en une liste ordinaire avec les mêmes éléments.
- ...

Python

array vs list

- list peut contenir des éléments de types différents mais pas array.
- list est généralement plus grande en termes d'utilisation de la mémoire pour stocker le même nombre d'éléments que array.
- list offre un large éventail de méthodes qui permettent des opérations telles que l'ajout, la suppression et la modification.

Python

typing

- Introduit dans **Python 3.5**.
- Facilitant ainsi le typage statique.
- Fournissant des outils pour spécifier les types de variables et les retours de fonction.

Python

typing : vue d'ensemble

- Généricité : `TypeVar:` et `Generic[T]`
- Types spéciaux : `Union`, `Any`, `Optional[T]`, `NoReturn`...
- Valeurs autorisées : `Literal`
- ...

Python

Exemple avec `Literal`

```
from typing import Literal

def couleur_preferee(couleur: Literal['rouge', 'vert', 'bleu']):
    print(f"Ma couleur pr´ef´er´ee est {couleur}")

couleur_preferee('rouge')
# affche : Ma couleur pr´ef´er´ee est rouge

couleur_preferee('jaune')
# erreur avec MyPy
```

Python

Avec `typing` (**Python 3.5**), il est possible d'utiliser les itérables génériques suivants

- `List[]`
- `Tuple[]`
- `Dict[]`
- `Set[]`
- `FrozenSet[]`
- `Deque[]`

Python

Remarque

Depuis **Python 3.9**, il est possible d'utiliser les types built-in génériques de manière similaire aux types du module `typing`

- `list[]`
- `tuple[]`
- `dict[]`
- `set[]`
- `frozenSet[]`