

# Essentiels en python 2

**Sabrine BENDIMERAD : [sabrine.bendimerad1@gmail.com](mailto:sabrine.bendimerad1@gmail.com)**  
**Data science&AI Expert**

# Objectifs et Organisations

- Ce cours prépare à la certification [PCAP™ – Certified Associate in Python Programming](#)
- Documents utilisés :
- Références :
  - ❑ Cours de la Cisco Networking Academy
  - ❑ Cours en ligne :  
<https://openclassrooms.com/fr/courses/4302126-decouvrez-la-programmation-orientee-objet-avec-python> et <https://www.coursera.org/specializations/python>
  - ❑ Programming Python, 4th Edition, Publisher(s): O'Reilly Media, Inc.

# Déroulement de la séance

- Les exemples de code requièrent souvent des notions encore inconnues. Pas d'inquiétude si tout n'est pas clair à chaque fois !
- Evitez le mode «audience passive», n'hésitez pas à interagir avec moi et entre vous 😊.
- Suivez les exemples de code et testez sur vos environnements de travail !
- Il n'y a pas de question idiote, alors n'hésitez jamais !
- Ne soyez pas étonnés si je n'ai pas toujours les réponses à vos questions (Je regarderai et reviendrai vers vous )
- Vos retours/remarques/suggestions pour l'amélioration de ce cours sont les bienvenus en fin de séance.

# Plan du cours

- **Module 1** : Modules, packages, pip
- **Module 2** : Strings, méthodes de liste, exceptions
- **Module 3** : Programmation orientée objet
- **Module 4** : Generators, iterators, closures, file streams, processing text and binary files, the os, time, datetime, and calendar module

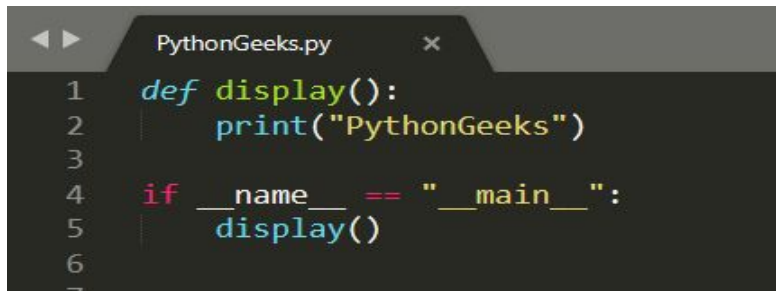
# MODULE 1

# Module 1 : Modules et packages

- Script Python
  - Un script Python est un fichier contenant du code Python exécutable.
  - Il est principalement utilisé pour effectuer des tâches spécifiques ou résoudre des problèmes simples.
- Module Python
  - Un module Python est un fichier Python (.py) contenant des fonctions, des variables et des classes réutilisables.
  - Les modules permettent de mieux organiser et répartir le code.
- Package Python
  - Un package Python est un dossier contenant plusieurs modules Python.
  - Les packages aident à structurer et à hiérarchiser le code, facilitant la gestion de projets plus importants.
  - Les packages Python sont des dossiers contenant plusieurs modules Python et un fichier `__init__.py` qui peut être vide mais reste nécessaire pour reconnaître un package.

# Module 1 : Modules et packages

- Créer un module :



```
PythonGeeks.py
1 def display():
2     print("PythonGeeks")
3
4 if __name__ == "__main__":
5     display()
6
7
```

Observez la fonction  
main qu'on expliquera  
plus tard !

- Importer un module :



```
#importer un module
import pythongeeks
pythongeeks.display()

from pythongeeks import *
display()

from pythongeeks import display
display()
```

# Module 1 : Modules et packages

- Créer un package:
  - Pour transformer notre module en package, on crée un repertoire par exemple “websites”
  - On ajoute un fichier `__init__.py`
- Importer un package :

```
import website.pythongeeks
pythongeeks.display()

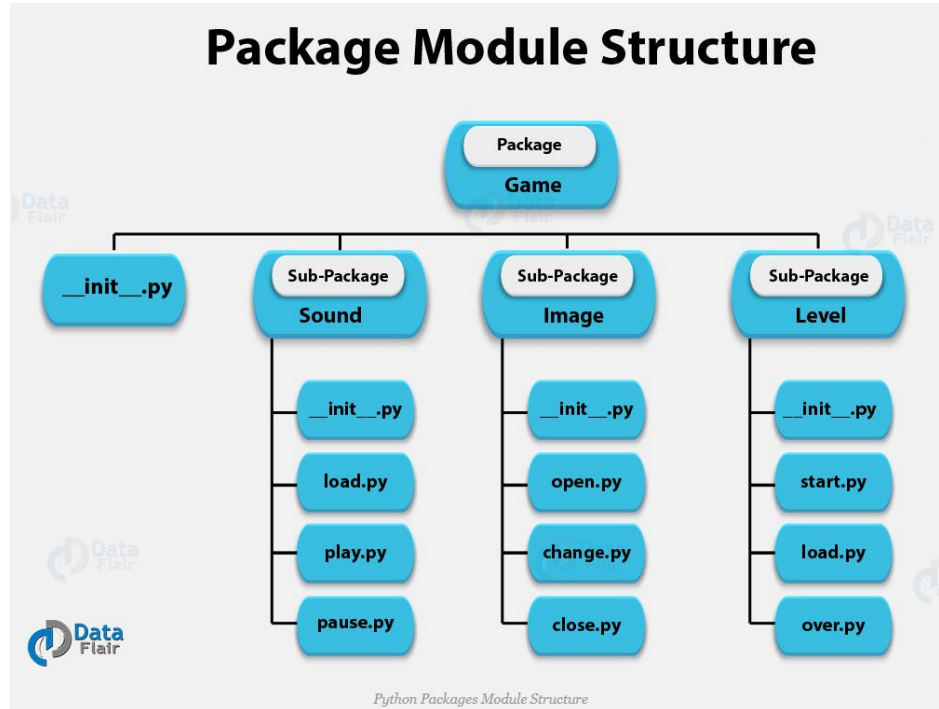
import website.pythongeeks as pg
pg.display()
Output
```

=> La fonction `main()` :

- En Python, la fonction `main()` est souvent utilisée comme point d'entrée de l'exécution d'un script.
- Elle est appelée automatiquement lorsque le script est exécuté.



# Module 1 : Modules et packages



# MODULE 2

# Module 2 : Méthodes pour chaînes de caractères

- `capitalize()` : Met toutes les lettres de la chaîne en majuscules.
- `center()` : Centre la chaîne dans un espace de longueur déterminée.
- `count()` : Compte les occurrences d'un caractère donné.
- `join()` : Joint tous les éléments d'un tuple/liste en une seule chaîne.
- `lower()` : Convertit toutes les lettres de la chaîne en minuscules.
- `lstrip()` : Supprime les espaces blancs du début de la chaîne.
- `replace()` : Remplace une sous-chaîne donnée par une autre.
- `rfind()` : Recherche une sous-chaîne en commençant par la fin de la chaîne.
- `rstrip()` : Supprime les espaces blancs de fin de chaîne.
- `split()` : Divise la chaîne en sous-chaînes en utilisant un délimiteur donné.
- `strip()` : Supprime les espaces blancs au début et à la fin de la chaîne.
- `swapcase()` : Inverse la casse des lettres (majuscules en minuscules et vice versa).
- `title()` : Met la première lettre de chaque mot en majuscule.
- `upper()` : Convertit toutes les lettres de la chaîne en majuscules

# Module 2 : Méthodes pour chaînes de caractères

- `endswith()` : La chaîne se termine-t-elle par une sous-chaîne donnée ?
- `isalnum()` : La chaîne contient-elle uniquement des lettres et des chiffres ?
- `isalpha()` : La chaîne contient-elle uniquement des lettres ?
- `islower()` : La chaîne contient-elle uniquement des lettres en minuscules ?
- `isspace()` : La chaîne contient-elle uniquement des espaces blancs ?
- `isupper()` : La chaîne contient-elle uniquement des lettres en majuscules ?
- `startswith()` : La chaîne commence-t-elle par une sous-chaîne donnée ?

# Module 2 : Crypter un message en python

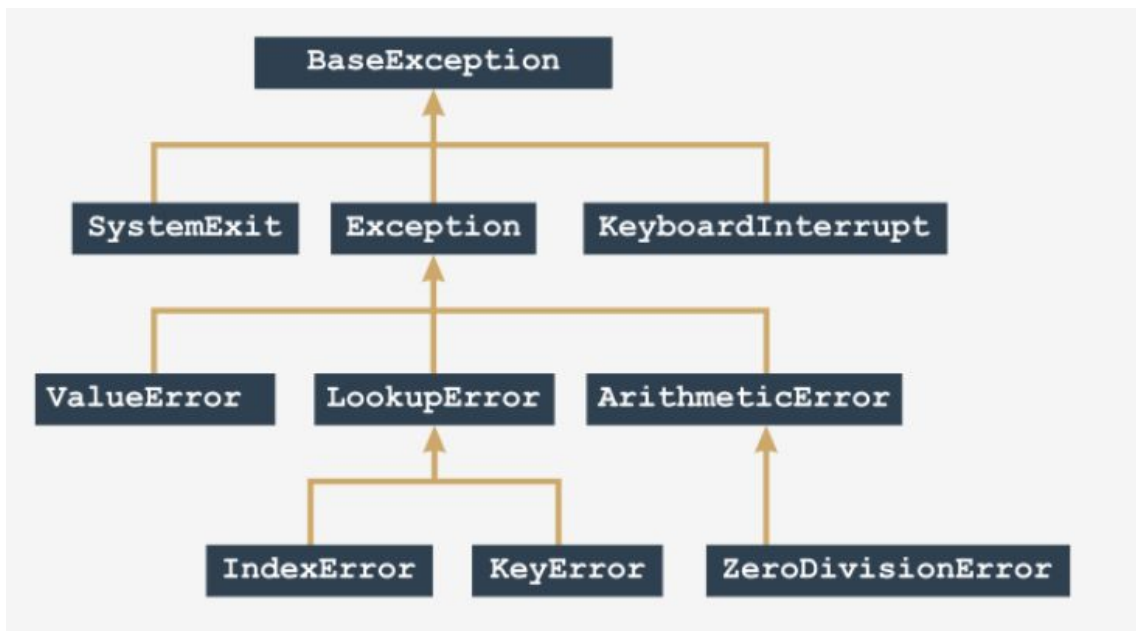
- Le chiffre de César est une technique de cryptage utilisée par Jules César et ses troupes pendant les guerres en Gaule.
- Chaque lettre du message est décalée d'une position vers la droite (A devient B, B devient C, etc.).
- La lettre Z devient A.
- Le programme Python suivant crypte un message en utilisant le chiffre de César.
- Explication :
  - L'utilisateur entre un message ouvert (non crypté) en une seule ligne.
  - Le programme parcourt chaque caractère du message.
  - Les caractères non alphabétiques sont ignorés.
  - Chaque lettre est convertie en majuscules (pour simplifier).
  - Le code ASCII de la lettre est incrémenté de 1.
  - Si le code dépasse Z, il est réinitialisé à A.
  - Les caractères cryptés sont ajoutés au message crypté.
  - Le message crypté est affiché à la fin.

## Module 2 : Crypter un message en python

```
# Caesar cipher.  
text = input("Enter your message: ")  
cipher = ''  
for char in text:  
    if not char.isalpha():  
        continue  
    char = char.upper()  
    code = ord(char) + 1  
    if code > ord('Z'):  
        code = ord('A')  
    cipher += chr(code)  
  
print(cipher)
```

# Module 2 : Les exceptions

- La gestion des exceptions est basée sur des blocs "try" et "except".
- Les exceptions peuvent être gérées à l'intérieur ou à l'extérieur des fonctions.



# Module 2 : Les exceptions

- Les exceptions sont des événements perturbant le déroulement normal du code.
- Python a une hiérarchie de 63 exceptions intégrées.
- La hiérarchie est organisée de manière arborescente, du général au spécifique.

```
try:
    y = 1 / 0
except ArithmeticError:
    print("Oooppsss...")

print("THE END.")

try:
    y = 1 / 0
except ZeroDivisionError:
    print("Zero Division!")
except ArithmeticError:
    print("Arithmetic problem!")

print("THE END.")
```

```
def bad_fun(n):
    try:
        return 1 / n
    except ArithmeticError:
        print("Arithmetic Problem!")
    return None

bad_fun(0)
print("THE END.")

def bad_fun(n):
    raise ZeroDivisionError

try:
    bad_fun(0)
except ArithmeticError:
    print("What happened? An error?")

print("THE END.")
```



# MODULE 3

# Module 3 : Classes et objets

- Méthode d'organisation des programmes via des objets
- Objets regroupent des propriétés (ex : nom, adresse, âge) et comportements (ex : marcher, parler)
- Les objets possèdent des données et exécutent des fonctions
- Développement plus rapide, coût réduit, meilleure maintenabilité
- Logiciel de meilleure qualité, extensible avec nouvelles méthodes et attributs
- Courbe d'apprentissage plus exigeante
- Complexité peut dérouter les débutants

# Module 3 : Classes et objets

- Une classe est une idée (plus ou moins abstraite) utilisée pour créer plusieurs incarnations - chaque incarnation est appelée un objets.
- Les objets sont équipés de :
  - Un nom pour les identifier et les distinguer ;
  - Un ensemble de propriétés (l'ensemble peut être vide) ;
  - Un ensemble de méthodes (qui peut également être vide).

```
class This_Is_A_Class:  
    pass  
  
this_is_an_object = This_Is_A_Class()
```

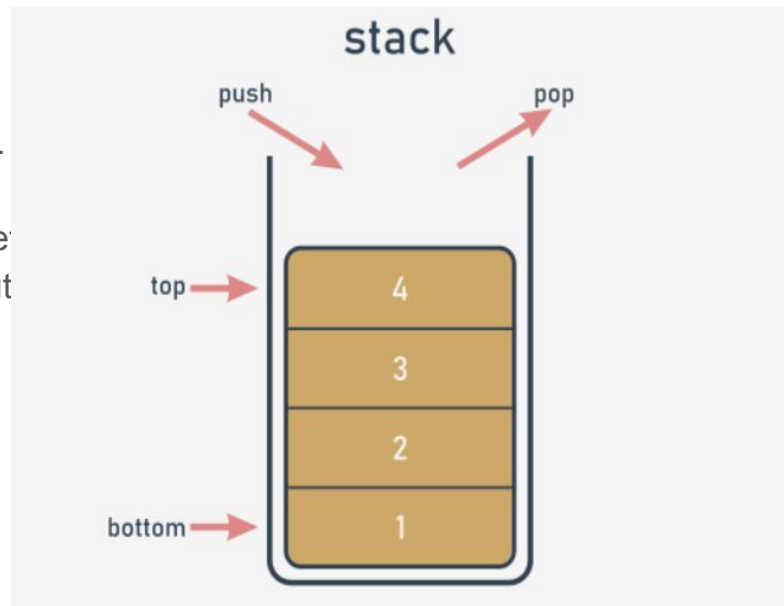
# Module 3 : Classes et objets

Pile (stack) :

- Structure de données pour stocker de manière spécifique.
- Comparée à une pile de pièces : ajouter/enlever en haut.
- Notion LIFO (Last In, First Out) : dernier ajouté, premier retiré.
- Opérations : "push" (ajout en haut) et "pop" (retrait du haut).
- Utilisée dans des algorithmes classiques.

Concept de la pile :

- Implémentation d'une pile en Python.
  - a. Deux approches : procédurale et orientée objet.
  - b. Commençons par la première approche.



# Module 3 : Classes et objets

- Utilisation d'une liste pour stocker les valeurs dans la pile.
- La pile est représentée par une liste vide :
  - Fonction push pour ajouter une valeur à la pile.
  - Prend un paramètre (la valeur à ajouter).
  - N'a pas de valeur de retour.
  - Ajoute la valeur à la fin de la pile.
- Fonction pop pour retirer une valeur de la pile.
  - Ne prend pas de paramètre.
  - Renvoie la valeur retirée de la pile.
  - Lit la valeur en haut de la pile et la supprime.

```
stack = []

def push(val):
    stack.append(val)

def pop():
    val = stack[-1]
    del stack[-1]
    return val

push(3)
push(2)
push(1)

print(pop())
print(pop())
print(pop())
```

# Module 3 : Classes et objets

Avantage de l'approche orientée objet pour la gestion de piles :

- Procédurale :
  - Nécessite des listes distinctes et des fonctions dupliquées pour chaque pile.
- Orientée objet :
  - Utilisation d'une classe Stack encapsulant la logique de gestion de pile.
  - Instanciation d'objets Stack pour représenter des piles distinctes.
  - Pas de duplication de code grâce à la réutilisation de la classe Stack.
- Possibilité d'étendre la classe Stack pour des fonctionnalités spécifiques à une pile.
- Résultat : Code plus propre, modulaire et facile à maintenir.

```
class Stack:
    def __init__(self):
        self.stack_list = []

stack_object_1 = Stack()
stack_object_2 = Stack()

print(len(stack_object1.stack_list))
print(len(stack_object1.stack_list))
```

# Module 3 : L'encapsulation

```
class Stack:
    def __init__(self):
        self.stack_list = []

stack_object_1 = Stack()
stack_object_2 = Stack()

print(len(stack_object1.stack_list))
print(len(stack_object1.stack_list))
```

```
class Stack:
    def __init__(self):
        self.__stack_list = []

stack_object_1 = Stack()
stack_object_2 = Stack()

print(len(stack_object1.__stack_list))
print(len(stack_object1.__stack_list))
```

Vous remarquez le double underscore ?

- Les composants d'une classe commençant par deux tirets bas (\_\_), deviennent privés.
- Les composants privés ne peuvent être accédés qu'à l'intérieur de la classe.
- Le concept d'encapsulation est mis en œuvre en restreignant l'accès externe.

# Module 3 : L'encapsulation

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val

stack_object_1 = Stack()
stack_object_2 = Stack()

stack_object_1.push(3)
stack_object_2.push(stack_object_1.pop())

print(stack_object_2.pop())
```

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val

stack_object = Stack()

stack_object.push(3)
stack_object.push(2)
stack_object.push(1)

print(stack_object.pop())
print(stack_object.pop())
print(stack_object.pop())
```



# Module 3 : L'héritage

- Création d'une nouvelle classe AddingStack.
  - Objectif : calculer la somme des éléments ajoutés à une stack.
- Pour cela :
  - Utilisation d'une sous-classe pour éviter d'altérer la classe principale.
  - La sous-classe hérite de la classe Stack.
  - Ajout d'une variable privée `__sum` pour stocker la somme.
  - Appel explicite du constructeur de la superclasse Stack dans le constructeur de la sous-classe AddingStack.

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val

class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0
```

- Contrairement à d'autres langages, Python exige l'invocation explicite du constructeur de la superclasse.

```
class Stack:
    def __init__(self):
        self.__stack_list = []

    def push(self, val):
        self.__stack_list.append(val)

    def pop(self):
        val = self.__stack_list[-1]
        del self.__stack_list[-1]
        return val

class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0

    def get_sum(self):
        return self.__sum

    def push(self, val):
        self.__sum += val
        Stack.push(self, val)

    def pop(self):
        val = Stack.pop(self)
        self.__sum -= val
        return val
```

```
stack_object = AddingStack()

for i in range(5):
    stack_object.push(i)
print(stack_object.get_sum())

for i in range(5):
    print(stack_object.pop())
```

# Module 3 : Variables d'instances

- Une classe peut être équipée de deux types de données pour former ses propriétés.
  - Propriétés créées explicitement et ajoutées à un objet.
  - Peut être fait lors de l'initialisation de l'objet ou à n'importe quel moment de sa durée de vie.
- Chaque objet possède son propre ensemble de propriétés - pas d'interférence.
- Remarques :
  - Les objets Python possèdent une variable prédéfinie nommée `__dict__`.
  - Elle contient les noms et valeurs de toutes les propriétés de l'objet.

```
class ExempleClasse:
    def __init__(self, val = 1):
        self.first = val

    def set_second(self, val):
        self.second = val

objet_1 = ExempleClasse()
objet_2 = ExempleClasse(2)
objet_2.set_second(3)
objet_3 = ExempleClasse(4)
objet_3.third = 5

print(objet_1.__dict__)
print(objet_2.__dict__)
print(objet_3.__dict__)
```

# Module 3 : Variables d'instances

- Afficher une instance de classe protégée :
  - Pour rendre une variable d'instance privée, ajouter deux tirets bas devant son nom (\_\_nom).
  - Vous pouvez accéder à la variable privée depuis l'extérieur en utilisant \_NomDeClasse\_\_nom.

```
class ExampleClass:
    def __init__(self, val = 1):
        self.__first = val

    def set_second(self, val = 2):
        self.__second = val

object_1 = ExampleClass()
object_2 = ExampleClass(2)

print(object_1._ExampleClass__first)
```

# Module 3 : Variables de classe

- Une variable de classe est une propriété qui existe en une seule copie et est stockée en dehors de tout objet.
- Une variable de classe existe même s'il n'y a pas d'objet dans la classe.
- Elles sont créées différemment des variables d'instance.

```
class ExampleClass:
    counter = 0
    def __init__(self, val = 1):
        self.__first = val
        ExampleClass.counter += 1

object_1 = ExampleClass()
object_2 = ExampleClass(2)
object_3 = ExampleClass(4)

print(object_1.__dict__, object_1.counter)
print(object_2.__dict__, object_2.counter)
print(object_3.__dict__, object_3.counter)
```

# Module 3 : Méthodes

- Une méthode est une fonction à l'intérieur d'une classe.
- Elle doit avoir au moins un paramètre (généralement nommé "self").
- "self" identifie l'objet pour lequel la méthode est invoquée.
- Utilisez "self" comme premier paramètre par convention.
- La méthode spéciale `__init__`, appelée constructeur est automatiquement appelée lors de la création d'un objet.

```
class MyClass:
    def regular_method(self, parameter):
        print("Regular method:", parameter)

    def __init__(self, value):
        self.initial_value = value
        print("Constructor called with value:", value)

obj = MyClass(5)

obj.regular_method("Hello")
```

# Module 3 : Méthodes spéciales

- `__name__` retourne le nom de la classe en tant que chaîne de caractères.
- `__module__` retourne le nom du module dans lequel la classe est définie.
- `__bases__` retourne un tuple contenant les classes de base directes de la classe. : Elle permet de savoir quelles classes sont héritées.

```
class SuperOne:
    pass

class SuperTwo:
    pass

class Sub(SuperOne, SuperTwo):
    pass

print(Sub.__name__)
print(Sub.__module__)
print(Sub.__bases__)
```

# Module 3 : Méthodes spéciales (\_\_str\_\_)

- La méthode \_\_str\_\_ permet de personnaliser la façon dont un objet est affiché avec la fonction print().
- Par défaut, print(objet) affiche une représentation peu lisible de l'objet.
- En définissant la méthode \_\_str\_\_ dans la classe, on peut améliorer la présentation de l'objet.

```
class Star:
    def __init__(self, name, galaxy):
        self.name = name
        self.galaxy = galaxy

sun = Star("Sun", "Milky Way")
print(sun)
```

```
class Star:
    def __init__(self, name, galaxy):
        self.name = name
        self.galaxy = galaxy

    def __str__(self):
        return self.name + ' in ' + self.galaxy

sun = Star("Sun", "Milky Way")
print(sun)
```



# Module 3 : L'héritage

- Un objet est une incarnation d'une classe, comme un gâteau cuit avec une recette contenue dans la classe.
- Lorsque vous avez un objet, vous pouvez vous demander à quelle recette (classe) il appartient.
- La fonction `isinstance(objet, classe)` permet de vérifier si un objet est une instance d'une classe donnée.

```
class Vehicule:
    pass

class VehiculeTerrestre(Vehicule):
    pass

class VehiculeAChenilles(VehiculeTerrestre):
    pass

for cls1 in [Vehicule, VehiculeTerrestre, VehiculeAChenilles]:
    for cls2 in [Vehicule, VehiculeTerrestre, VehiculeAChenilles]:
        print(isinstance(cls1, cls2), end="\t")
    print()
```

# Module 3 : Gestion des propriétés et méthodes

- Super est la superclasse qui contient :
  - Une méthode `__init__` pour initialiser
  - Un nom et une méthode `__str__` pour afficher le nom.
- Sub est la sous-classe qui hérite de Super.
- Sub utilise `super()` pour appeler la méthode `__init__` de la superclasse.
- Lorsque nous créons un objet Sub et imprimons, la méthode `__str__` héritée de Super est utilisée pour afficher le résultat.

```
class Super:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return "My name is " + self.name + "."

class Sub(Super):
    def __init__(self, name):
        # Super.__init__(self, name)
        super().__init__(name)

obj = Sub("Andy")
print(obj)
```

# Module 3 : Propriétés et méthodes

- Lorsqu'on accède à une propriété ou à une méthode d'un objet en Python, le langage suit un ordre précis :
  - Il recherche d'abord dans l'objet lui-même.
  - Il recherche dans les classes impliquées dans la chaîne d'héritage de l'objet, de bas en haut.
  - Si plusieurs classes sont sur le même chemin d'héritage, Python les explore de gauche à droite.

```
obj = Level3()
print(obj.variable_1, obj.var_1, obj.fun_1())
print(obj.variable_2, obj.var_2, obj.fun_2())
print(obj.variable_3, obj.var_3, obj.fun_3())
```

```
class Level1:
    variable_1 = 100
    def __init__(self):
        self.var_1 = 101

    def fun_1(self):
        return 102

class Level2(Level1):
    variable_2 = 200
    def __init__(self):
        super().__init__()
        self.var_2 = 201

    def fun_2(self):
        return 202

class Level3(Level2):
    variable_3 = 300
    def __init__(self):
        super().__init__()
        self.var_3 = 301

    def fun_3(self):
        return 302
```

# Module 3 : Propriétés et méthodes

- L'entité définie dans une classe enfant peut remplacer celle d'une classe parente.
- Python recherche les entités d'abord dans la classe elle-même, puis dans les classes parentes.

```
class Left:
    var = "L"
    var_left = "LL"
    def fun(self):
        return "Left"

class Right:
    var = "R"
    var_right = "RR"
    def fun(self):
        return "Right"

class Sub(Left, Right):
    pass

obj = Sub()

print(obj.var, obj.var_left, obj.var_right, obj.fun())
```