FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

DÉPARTEMENT DE MATHÉMATIQUES ET INFORMATIQUE

PROGRAMMATION PYTHON

(PYTHON et OPTIMISATION NUMERIQUE)

Master : MNSA

Pr: Youssef Ouassit

Objectifs du module

- Connaître les bases du langage Python
- Apprendre la programmation orientée objet en Python.
- Implémenter quelques exemples des méthodes d'optimisation
- Manipulation des librairies d’optimisation en Python.

Descriptif horaire

- 18 heures de Cours/TD;
- 6 heures de présentations :
  - Apprentissage automatique
  - Les heuristiques et méta-heuristiques
  - ….

Evaluation

Note finale du module Programmation Python:

- 30% Présentation
- 70% Examen écrit

FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

DÉPARTEMENT DE MATHÉMATIQUES ET INFORMATIQUE

Introduction à Python : Présentation de Python, La syntaxe de base

Master : MNSA

Pr: Youssef Ouassit

Un peu d’histoire

Origines (1980s - 1990)

Créateur : Python a été créé par **Guido van Rossum**, un programmeur néerlandais, à la fin des années 1980. Van Rossum travaillait au Centrum Wiskunde & Informatica (CWI) aux Pays-Bas lorsqu'il a commencé à développer Python.



Nom : Le nom "Python" ne vient pas du serpent, mais plutôt du groupe comique britannique Monty Python. Van Rossum voulait un nom court, unique et un peu mystérieux.

Un peu d’histoire

La version 0.9.0 (1991)

Première version publique : Python 0.9.0 a été publié en février 1991. Cette version incluait déjà des fonctionnalités clés telles que les classes avec héritage, les exceptions, et les fonctions avec arguments nommés. Les structures de données intégrées comme les listes, les dictionnaires et les chaînes de caractères étaient également présentes.

Un peu d’histoire

La version 1.0 (1994)

Python 1.0 a été publié en janvier 1994. Cette version a introduit des fonctionnalités importantes comme les modules, les expressions **lambda**, les fonctions **map**, **filter**, et **reduce**.

Adoption : À cette époque, Python a commencé à gagner en popularité grâce à sa syntaxe simple et à sa capacité à s'intégrer facilement à d'autres langages et outils.

Un peu d'histoire

La version 2.x (2000)

Python 2.0 a été publié en octobre 2000. Cette version a introduit de nombreuses améliorations, notamment la collecte des ordures (**garbage collection**) pour gérer la mémoire, la **liste des compréhensions**, et le support complet de **Unicode**.

Python 2.x a continué à évoluer avec de nombreuses versions mineures, ajoutant des fonctionnalités et améliorant la performance. Python 2.7, publié en 2010, est devenu la version finale de la série Python 2.

Un peu d’histoire

La version 3.x (2008 à présent)

Python 3.0 a été publié en décembre 2008. Cette version a introduit des changements majeurs qui ont **rompu** la compatibilité avec Python 2.x, rendant la transition difficile pour certains projets.
L'objectif principal de Python 3 était de corriger les défauts fondamentaux du langage, en rendant la syntaxe plus cohérente et en supprimant les fonctionnalités obsolètes.

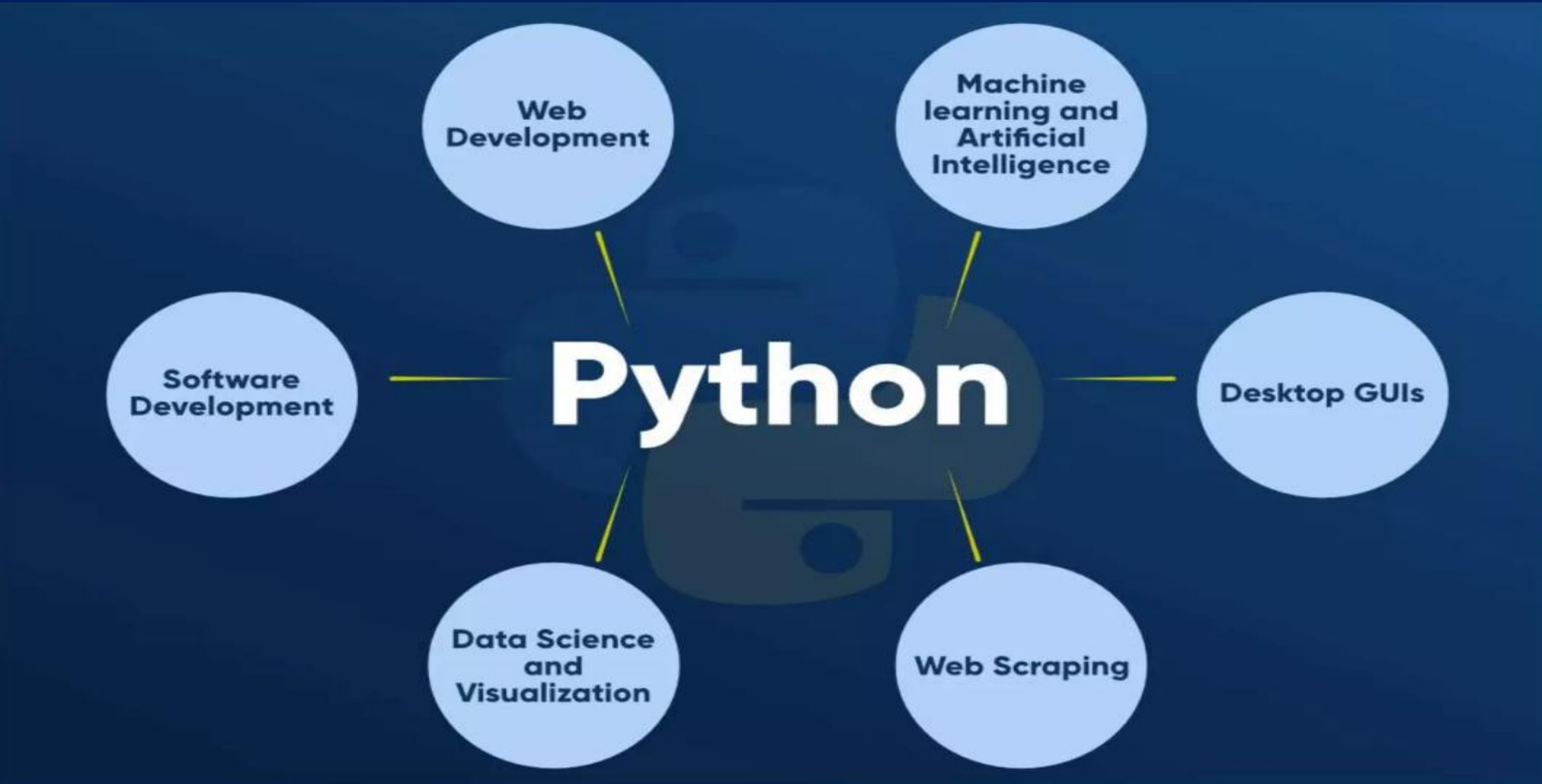
Adoption de Python 3 : Bien que la transition ait été lente au départ, Python 3 est maintenant la version principale utilisée par la majorité de la communauté Python. Le support officiel de Python 2 a pris fin le 1er janvier 2020, marquant la fin d'une ère.

Un peu d’histoire

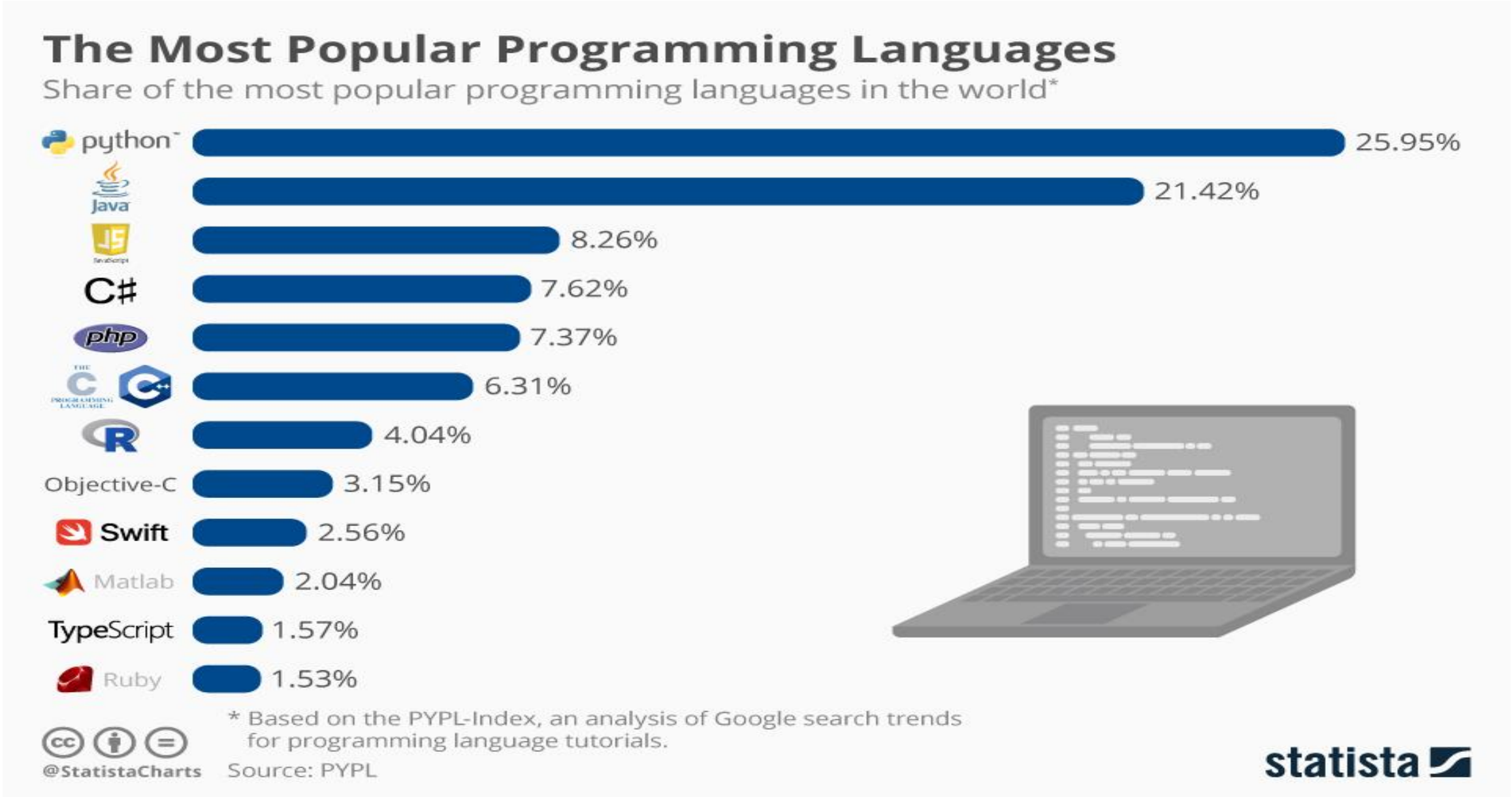
Python 3.x :

- Support de l'Unicode (sans utiliser le préfixe 'u')
- Division flottante par défaut pour les opérateurs de division
- print est devenu une fonction
- Meilleure gestion des exceptions
- Annotation de type (à partir de 3.5)
- *range() est devenu un itérable par défaut (au lieu de générer une liste en 2.x)*
- …

Domaines d’applications



Un peu d'histoire



Pourquoi Python ?

Caractéristiques de Python:

Simple: avec une syntaxe qui privilège la lisibilité, libéré de celle de C/C++ :

- Programme compact et lisible
- Pas de notions complexe (pointeurs, …)

Portable : disponible sous toutes les plateformes (Unix, Windows, …)

Open source: Supervisées par la Python Software Foundation (PSF)

Pourquoi Python ?

Riche: il incorpore plusieurs possibilités de langage:

tiré de la **programmation impérative** : séquences des instructions qui change l'état du système, structure de contrôle, manipulation de nombres comme les flottants, doubles, complexe, de structures complexes comme les tableaux, les dictionnaires, etc.

tiré des **langages de script** : accès au système, manipulation de processus, de l'arborescence fichier, d'expressions rationnelles, etc.

tiré de la **programmation fonctionnelle** : les fonctions sont dites «fonction de première classe», car elles peuvent être fournies comme argument d'une autre fonction, il dispose aussi de lambda expression, de générateur etc.

tiré de la **programmation orienté objet** : définition de classe, héritage multiple, introspection (consultation du type, des méthodes proposées), ajout/retrait dynamique de classes, de méthode, compilation dynamique de code, délégation ("duck typing"), passivation/activation, surcharge d'opérateurs, etc.

Pourquoi Python, ses caractéristiques ?

Il est :

- **dynamique** : il n'est pas nécessaire de déclarer le type d'une variable dans le source. Le type est associé lors de l'exécution du programme ;

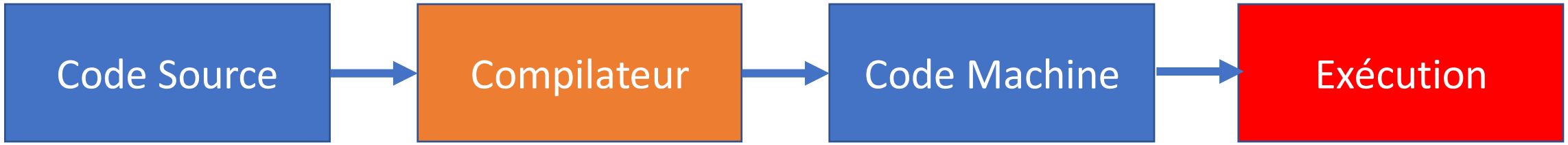
- **fortement typé** : les types sont toujours appliqués (un entier ne peut être considéré comme une chaîne sans conversion explicite, une variable possède un type lors de son affectation).

- **compilé/interprété** à la manière de Java. Le source est compilé en bytecode (pouvant être sauvegardé) puis exécuté sur une machine virtuelle.

Pourquoi Python, ses caractéristiques ?

Compilé vs Pseudo-Compilé vs interprété

Un langage compilé:



- Le code source est directement traduit en code machine natif spécifique à une plateforme.
- Exécution rapide car tout est compilé en amont.
- Nécessite une recompilation pour chaque type de machine.
- Exemple : C, C++

Pourquoi Python, ses caractéristiques ?

Compilé vs Pseudo-Compilé vs interprété

Un langage interprété:

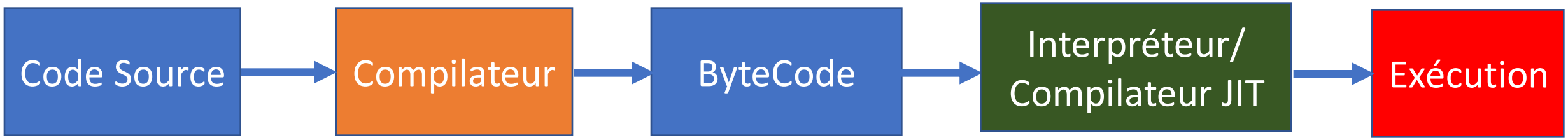


- Le code source est interprété ligne par ligne à l'exécution.
- Facilité de développement et de debugging, mais exécution plus lente.
- Aucune compilation préalable en code machine.

Pourquoi Python, ses caractéristiques ?

Compilé vs Pseudo-Compilé vs interprété

Un langage Pseudo-Compilé:

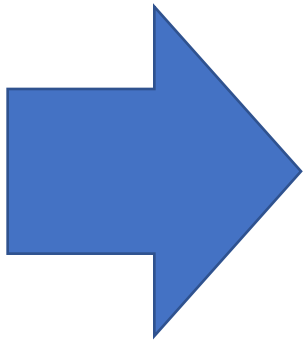


- Le code source est d'abord compilé en bytecode indépendant de la plateforme.
- Le bytecode est ensuite interprété ou compilé en code machine par la VM lors de l'exécution.
- Portabilité élevée car le bytecode peut être exécuté sur toute machine avec une VM.

Pourquoi Python, ses caractéristiques ?

Compilé vs Pseudo-Compilé vs interprété

Python c'est quoi ?



Python est principalement considéré comme un langage interprété, mais il inclut une étape intermédiaire de compilation en bytecode avant que l'interpréteur ne prenne le relais. C'est pour cela qu'on dit que Python est pseudo-compilé ou qu'il combine des éléments des langages compilés et interprétés.

Pourquoi Python, ses caractéristiques ?

Compilé vs Pseudo-Compilé vs interprété

Un langage Pseudo-Compilé:

```
import dis

def add(a, b):
    return a + b

# Désassemblage du bytecode
dis.dis(add)
```

```
2           0 LOAD_FAST                0 (a)
            2 LOAD_FAST                1 (b)
            4 BINARY_ADD
            6 RETURN_VALUE
```

Plusieurs implémentation de Python

La spécification VS l'implémentation :

Python le langage (la spécification) : Python est un langage de programmation de haut niveau, interprété.

C'est une spécification du langage, ce qui signifie qu'il définit la syntaxe, la sémantique, et le comportement de base que toute implémentation de Python doit suivre.

A travers : Les PEP (Python Enhancement Proposals)

Plusieurs implémentation de Python

La spécification VS l'implémentation :

Les implémentations de Python:

- ***CPython*** : L'implémentation standard de Python, écrit en langage C.
- ***PyPy***: Une implémentation de Python avec un compilateur Just-In-Time (JIT), qui peut rendre l'exécution du code Python plus rapide que CPython.
- ***Jython*** : Une implémentation de Python qui s'exécute sur la machine virtuelle Java (JVM). Elle permet au code Python d'interagir avec les bibliothèques Java.
- ***IronPython*** : Une implémentation de Python pour le framework .NET, permettant d'utiliser Python avec les langages et bibliothèques .NET.
- ***MicroPython*** : Une implémentation légère de Python conçue pour les microcontrôleurs et les environnements contraints.

Plusieurs implémentation de Python

La spécification VS l'implémentation :

Choisir l'implémentation:

```
$ python script.py
```

```
$ pypy script.py
```

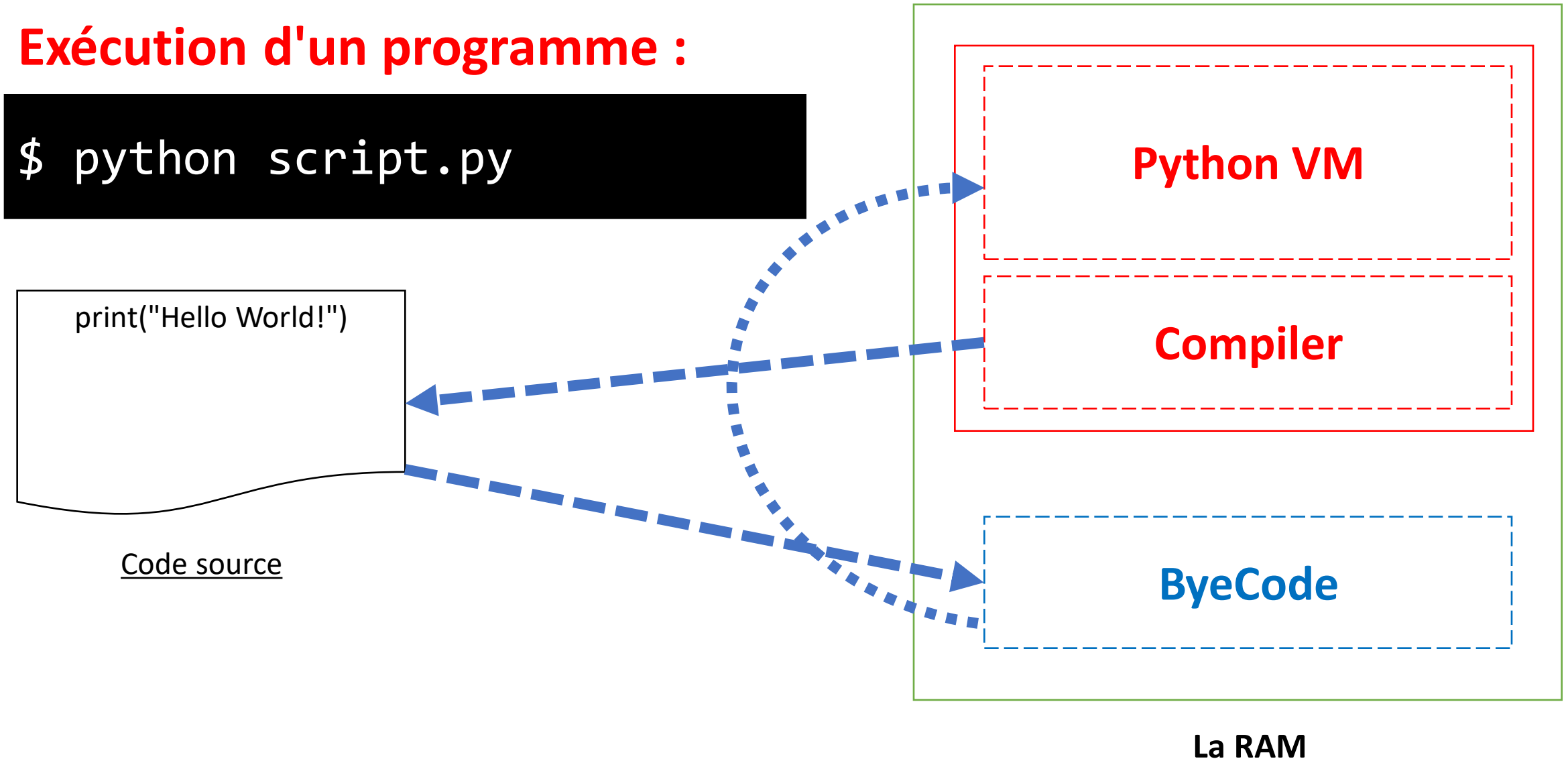
```
$ jpython script.py
```

```
$ ipy script.py
```

Plusieurs implémentation de Python

Exécution d'un programme :

```
$ python script.py
```



Installation de Python

Installation sur Windows:

Téléchargement : Rendez-vous sur le site officiel de Python www.python.org et téléchargez la dernière version stable.

Exécution de l'Installateur : Lancez l'installateur. Assurez-vous de cocher la case "Add Python to PATH" avant de cliquer sur "Install Now".

Vérification de l'Installation : Ouvrez l'invite de commande et tapez python --version ou python -V pour vérifier que Python est installé correctement.

Installation de Python

Installation sur Windows: Installation via Anaconda

Téléchargement d'Anaconda : Anaconda est une distribution Python qui inclut de nombreuses bibliothèques scientifiques et outils de gestion d'environnement. Téléchargez-le depuis www.anaconda.com .

Installation : Suivez les instructions spécifiques à votre système d'exploitation.

Vérification de l'Installation : Ouvrez Anaconda Prompt ou le terminal et tapez conda --version pour vérifier que Anaconda est installé.

Installation de Python













Jupyter:

- **Idéal pour la data science** : Jupyter est un environnement interactif parfait pour la visualisation et l'analyse de données, très utilisé en data science, machine learning, et dans le monde académique.

- **Exécution cellulaire** : Il permet d'exécuter des cellules de code indépendamment les unes des autres, ce qui est pratique pour tester des blocs de code sans exécuter l'ensemble du programme.

- **Visualisation directe** : Intègre des bibliothèques comme Matplotlib et Seaborn pour des graphiques interactifs et des analyses en temps réel.

- **Collaboration** : Possible de partager votre document en ligne.

Mode d'Emploi de l'Interpréteur Python :

Exécution de Scripts :

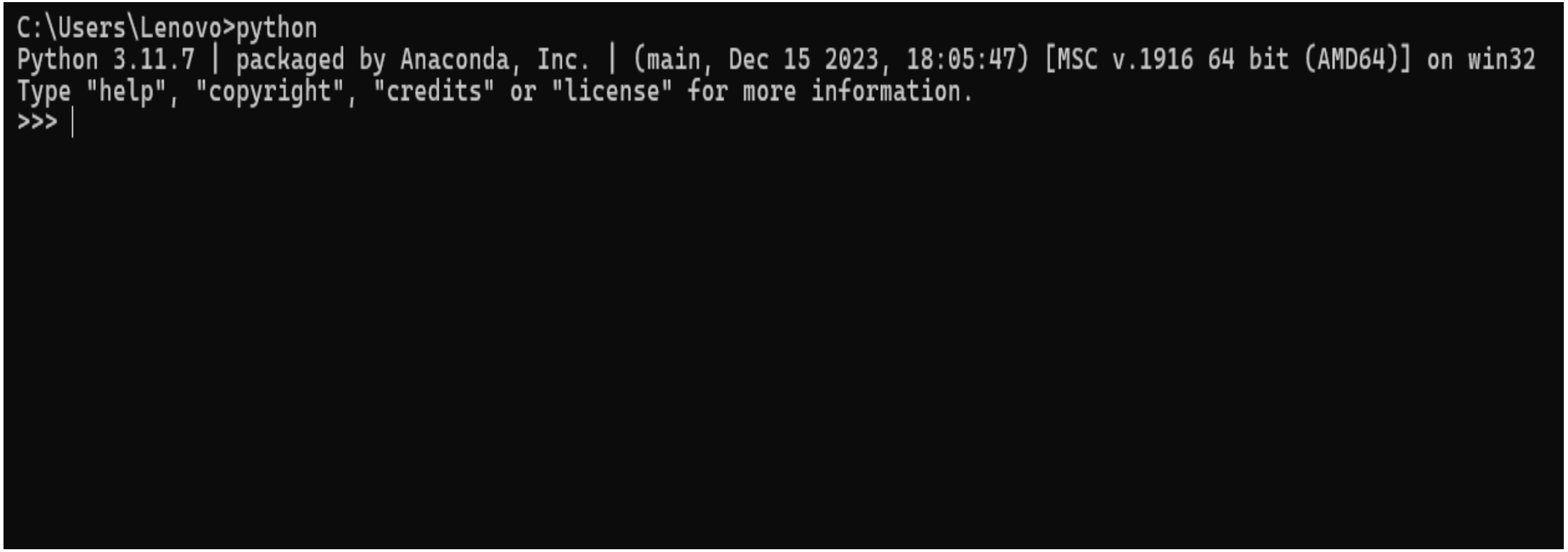
```
python script.py
```

Mode interactif:

En lançant simplement **python** ou **python3**, vous entrez dans le mode interactif. Cela vous permet d'exécuter des commandes Python ligne par ligne.

Mode d'Emploi de l'Interpréteur Python :

Mode interactif:



Syntaxe de base

Python est un langage de programmation interprété. C'est un «vrai» langage c.-à-d. types de données, branchements conditionnels, boucles, organisation du code en procédures et fonctions, objets et classes, découpage en modules.

Très bien structuré, facile à appréhender, c'est un langage privilégié pour l'enseignement.

Syntaxe de base

Les types de base :

int : Représente les nombres entiers, positifs ou négatifs, sans décimale.
Exemples : 5, -10, 0
Taille à partir de : 28 octets

float : Représente les nombres à virgule flottante, c'est-à-dire les nombres décimaux.
Exemples : 3.14, -0.001, 2.0
Taille à partir de : 24 octets

Syntaxe de base

Les types de base :

bool : Représente les valeurs de vérité, soit True soit False.
Exemples : True, False
Taille : 28 octets

complex : Représente les nombres complexes, qui ont une partie réelle et une partie imaginaire.
Exemples : 1 + 2j, 3 - 4j
Taille à partir de : 32 octets

Syntaxe de base

Les structure de données:

str : Représente une chaîne de caractères, utilisée pour stocker du texte.

list : Représente une collection ordonnée et modifiable d'éléments, qui peut contenir des éléments de différents types.

tuple : Similaire à une liste, mais immuable (les éléments ne peuvent pas être modifiés après la création).

range : Représente une séquence d'entiers, souvent utilisée dans les boucles.

dict : Représente une collection non ordonnée de paires clé-valeur. Les clés doivent être uniques et immuables, mais les valeurs peuvent être de n'importe quel type.

set : Représente une collection non ordonnée d'éléments uniques, sans doublons.

Syntaxe de base – Instructions de base

Affectation : Typage dynamique

Déclaration par affectation :

```
>> a = 1.2
```

a est une variable, en interne elle a été automatiquement typée en flottant «float» parce qu'il y a un point décimal. a est l'identifiant de la variable (attention à ne pas utiliser le mots réservés comme identifiant), = est l'opérateur d'affectation

Après un calcul :

```
>> a = 1.2
>> d = a + 3
```

d sera un réel contenant la valeur 4.2

Supprimer un objet de la mémoire

➢ **`del`** `nom_de_variable`

où nom_de_variable représente le nom de l'objet à supprimer.

Syntaxe de base – Instructions de base

Affectation : Typage dynamique

Forcer le typage d'une variable (sert aussi pour le transtypage)

```
>> b = float(1)
```

Même sans point décimal, b sera considéré comme float (b = 1, il aurait été int dans ce cas).

Connaître le type d'un objet

```
>> type(nom_de_variable)
```

Affiche le type interne d'une variable (ex. type(a) → <class 'float'>)

Concepts de base

Transtypage :

En Python, le **transtypage** (ou "conversion de types") désigne le processus de conversion d'une valeur d'un type de données à un autre. Il existe deux types :

1 - Transtypage implicite :

Python effectue automatiquement certaines conversions de type lorsqu'il est nécessaire de combiner différents types de données.

```
x = 10    # int
y = 3.5   # float
z = x + y # Python convertit 'x' en float avant l'addition
print(z)  # 13.5
```

Concepts de base

Transtypage :

En Python, le **transtypage** (ou "conversion de types") désigne le processus de conversion d'une valeur d'un type de données à un autre. Il existe deux types :

2 - Transtypage explicite :

Dans d'autres situations, il est nécessaire de convertir manuellement un type de données en un autre à l'aide de fonctions de conversion.

Principe Utilisation du mot-clé désignant le type

>> nouveau_objet = **nouveau_type** (objet)

Concepts de base

Transtypage :

Conversion en numérique

```
a = "12 " # a est de type chaîne caractère
b = int(a) #b est de type int
c = float(a) #c est de type float
```

N.B. Si la conversion n'est pas possible ex. float(« toto »), Python renvoie une erreur

Conversion en logique

```
a = bool("TRUE") # a est de type bool est contient la valeur True
a = bool(1) # renvoie True également
```

Conversion en chaîne de caractères

```
a = str(15) # a est de type chaîne et contient « 15 »
```

Concepts de base

Enchaînement des instructions

La plus couramment utilisé

1 instruction = 1 ligne

```
a = 1
b = 5
d = a + b
```

Autres possibilités

```
a = 1;b = 5 ;d = a + b;
```

Peu utilisé

```
a = 1;
b = 5;
d = a + b;
```

Concepts de base

La gestion de la mémoire :

les variables sont considérées comme des abstractions de haut niveau. Cela signifie qu'une variable en Python n'est pas directement liée à un emplacement mémoire fixe (comme dans les langages de bas niveau tels que C ou C++), mais qu'elle sert plutôt de référence à un objet qui est géré par l'interpréteur Python.

```
>>> a = 100
```

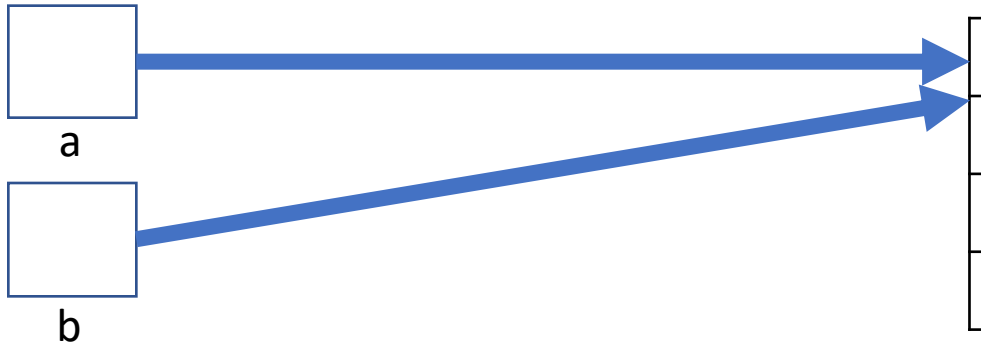


PyObject	
Type	int
Reference count	1
Value	100

Concepts de base

La gestion de la mémoire : Le compteur de références

```
>>> a = 100
>>> b = 100
```



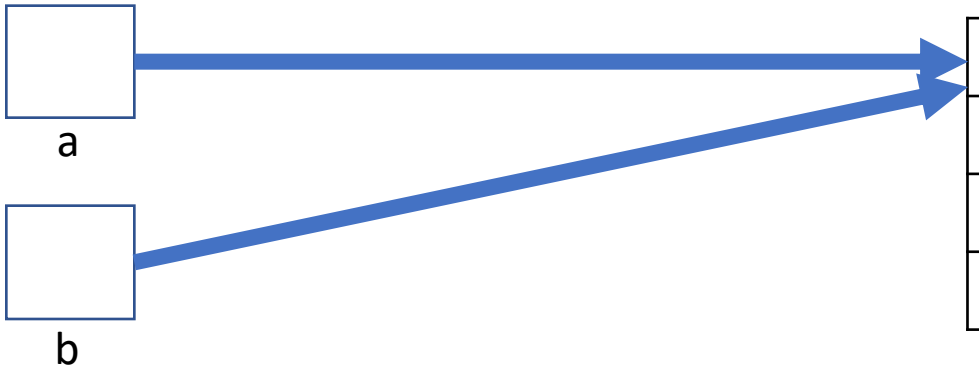
PyObject	
Type	int
Reference count	2
Value	100

```
>>> hex(id(a))
0x012493ff
>>> hex(id(b))
0x012493ff
>>> import sys;  sys.getrefcount(100)
2
```

Concepts de base

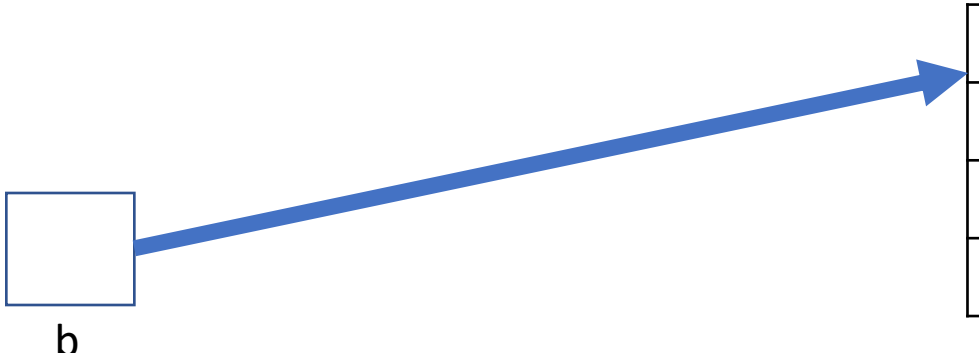
La gestion de la mémoire : Le compteur de références

```
>>> a = 500
>>> b = 500
```



PyObject	
Type	int
Reference count	2
Value	500

```
>>> del a
```

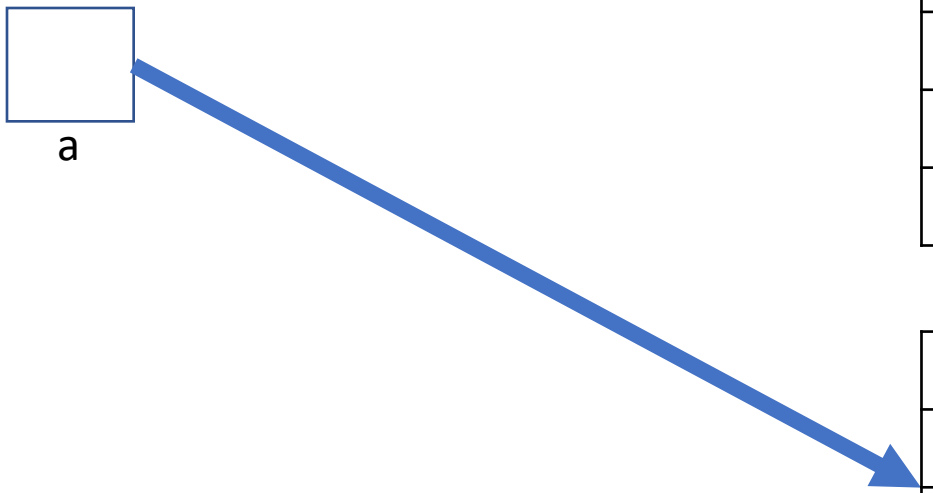


PyObject	
Type	int
Reference count	1
Value	500

Concepts de base

La gestion de la mémoire :

```
>>> a = 100
>>> a = 200
```



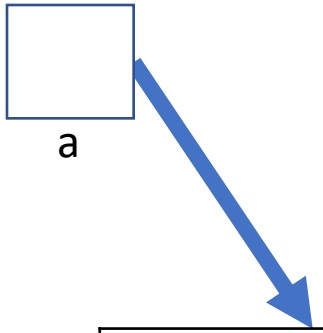
PyObject	
Type	int
Reference count	0
Value	100

PyObject	
Type	Int
Reference count	1
Value	200

Concepts de base

La gestion de la mémoire :

```
>>> a = 100
>>> a = 200
>>> a = "Hello"
```



PyObject	
Type	int
Reference count	0
Value	100

PyObject	
Type	str
Reference count	1
Value	Hello

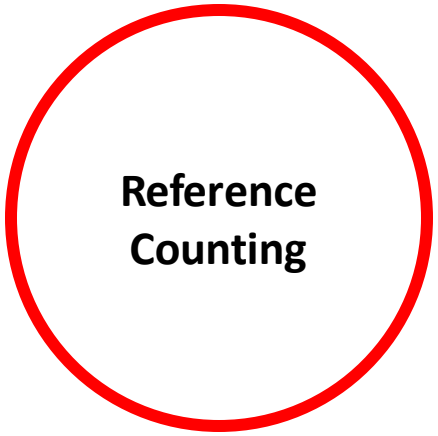
PyObject	
Type	Int
Reference count	0
Value	200

Concepts de base

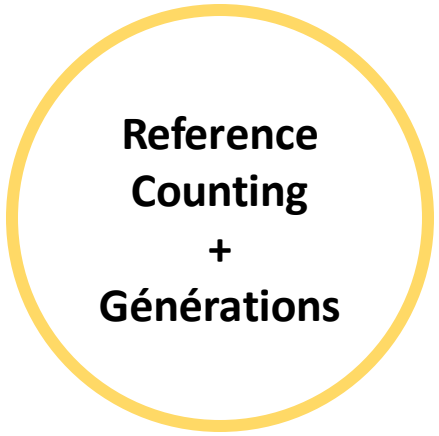
La gestion de la mémoire : Garbage Collector

En Python, le garbage collection (GC) est le processus de gestion automatique de la mémoire en identifiant et en libérant la mémoire qui n'est plus utilisée, c'est-à-dire les objets qui ne sont plus référencés nulle part dans le programme. Python utilise une combinaison de comptage de références et de garbage collection cyclique pour gérer la mémoire.

Trois stratégies principales :







Concepts de base

La gestion de la mémoire : Interning pour l'optimisations

Pour optimiser l'accès et la création de certains objets, Python interne les valeurs fréquemment utilisés. Pour le cas des nombre entier, par défaut, tous les entiers de -5 à 256 sont internés.
Cela signifie que ces valeurs sont stockées en mémoire à un seul endroit, et lorsque vous créez plusieurs variables avec la même valeur dans cette plage, elles pointeront vers le même objet en mémoire.

```
>>> sys.getrefcount(5)
>>> 1000000059
>>> sys.getrefcount(5000)
>>> 2
```

Concepts de base

La gestion de la mémoire :

```
>>> a = "Hello Ahmed"
>>> b = "Hello Ahmed"
>>> id(a), id(b)
>>> (2498768485296, 2498768486000)
>>> a = "Hello_Ahmed"
>>> b = "Hello_Ahmed"
>>> id(a), id(b)
>>> (2498694187120, 2498694187120)
```

Remarque : Si ces instructions sont exécuté à partir d'un script, les chaînes identiques auront le même id.

Concepts de base

Opérateurs arithmétiques:

Ces opérateurs sont utilisés pour effectuer des opérations mathématiques.

Addition (+) : Additionne deux valeurs.
Soustraction (-) : Soustrait la seconde valeur de la première.
Multiplication (*) : Multiplie deux valeurs.
Division (/) : Divise la première valeur par la seconde, renvoie un flottant.
Division entière (//) : Divise la première valeur par la seconde et renvoie un entier (partie entière).
Modulo (%) : Renvoie le reste de la division de la première valeur par la seconde.
Exponentiation (**) : Élève la première valeur à la puissance de la seconde.

Concepts de base

Opérateurs de comparaison:

Ces opérateurs comparent deux valeurs et renvoient un booléen (True ou False).

Égal à (==) : Vérifie si les deux valeurs sont égales.
Différent de (!=) : Vérifie si les deux valeurs sont différentes.
Supérieur à (>) : Vérifie si la première valeur est supérieure à la seconde.
Inférieur à (<) : Vérifie si la première valeur est inférieure à la seconde.
Supérieur ou égal à (>=) : Vérifie si la première valeur est supérieure ou égale à la seconde.
Inférieur ou égal à (<=) : Vérifie si la première valeur est inférieure ou égale à la seconde.

Concepts de base

Opérateurs logiques:

Ces opérateurs sont utilisés pour combiner des expressions conditionnelles.

Et (and) : Renvoie True si les deux conditions sont vraies.
Ou (or) : Renvoie True si au moins une des conditions est vraie.
Non (not) : Inverse la valeur de vérité.

Concepts de base

Opérateurs d'identité:

Ces opérateurs comparent l'identité des objets, c'est-à-dire s'ils sont le même objet en mémoire.

Est (is) : Renvoie True si les deux variables pointent vers le même objet.
N'est pas (is not) : Renvoie True si les deux variables pointent vers des objets différents.

Concepts de base

Opérateurs d'appartenance:

Ces opérateurs testent l'appartenance d'une valeur à une séquence (comme une liste, un tuple, une chaîne, etc.).

in : Renvoie True si la valeur est trouvée dans la séquence.

not in : Renvoie True si la valeur n'est pas trouvée dans la séquence.

Concepts de base

Opérateurs bit à bit:

Ces opérateurs travaillent directement sur les bits des entiers.

Et bit à bit (&) : Effectue un ET logique sur chaque bit des deux nombres.
Ou bit à bit (|) : Effectue un OU logique sur chaque bit des deux nombres.
Ou exclusif bit à bit (^) : Effectue un OU EXCLUSIF logique sur chaque bit des deux nombres.
Non bit à bit (~) : Inverse tous les bits du nombre.
Décalage à gauche (<<) : Décale les bits du nombre vers la gauche.
Décalage à droite (>>) : Décale les bits du nombre vers la droite.

Entré / Sortie

Instruction de lecture :

```
# syntaxe de l'instruction de lecture
v = input(message)
```

input() permet d'effectuer une saisie au clavier
Il est possible d'afficher un message d'invite.
La fonction renvoie toujours une chaîne, il faut convertir

input() collecte toujours une chaîne de caractères.

Exemple:

```
v = input("Votre nom:")
```

Entré / Sortie

Instruction de lecture :

Pour récupérer un entier :

```
a = input("Votre age:")
a = int(a)
```

Pour récupérer un flottant :

```
b = input("Le prix de l'article:")
b = float(b)
```

Combiner les deux fonctions :

```
a = int(input("Votre age:"))
b = float(input("Le prix de l'article:"))
```

Entré / Sortie

Instruction d'écriture:

```
# syntaxe de l’instruction de sortie
print(expression1, expression2, ...)
```

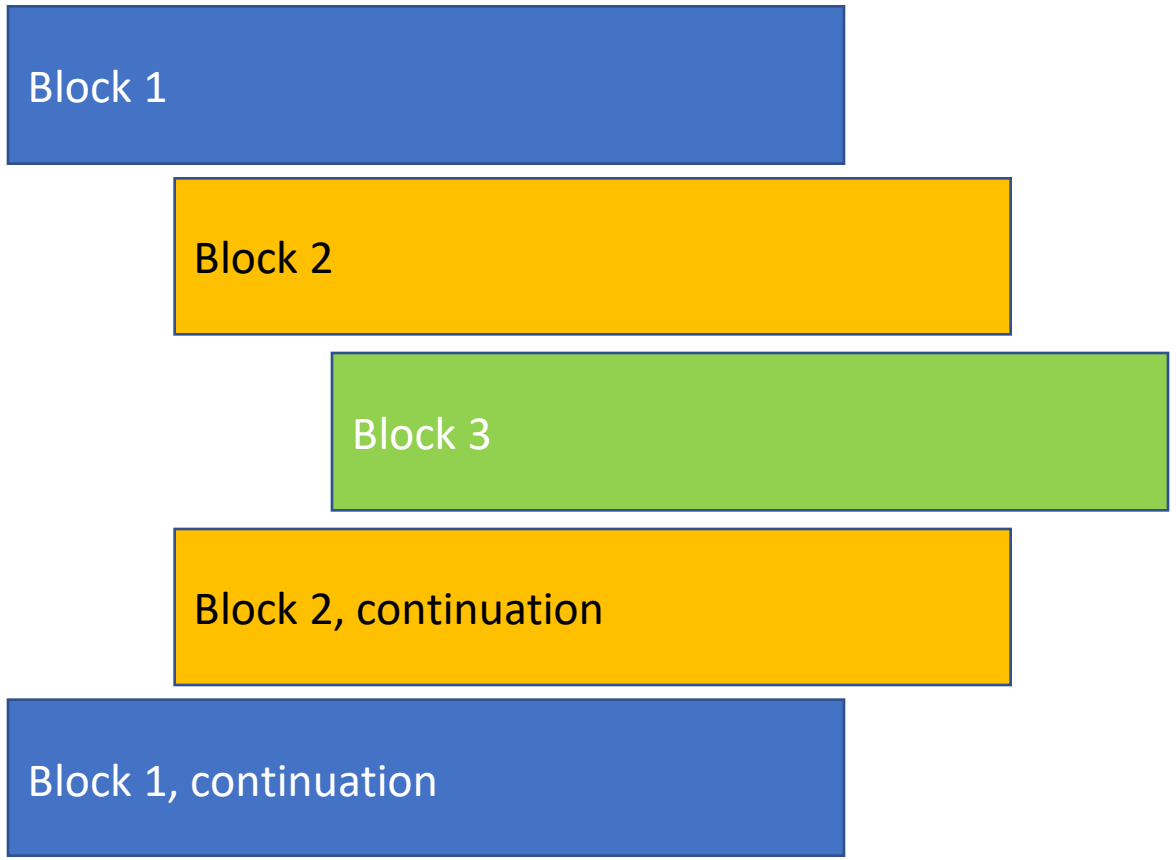
Exemple:

```
>>> a = 5
>>> print(a)
>>> print("La valeur est: ", a)
>>> print(f"La valeur est: {a}")
```

- L’affichage direct du contenu d’un tableau (liste) est possible également.

Concepts de base

La notion d'un bloque :



Indentation :

- Python utilise l'indentation (généralement 4 espaces) pour délimiter les blocs de code.
- Chaque ligne de code appartenant à un même bloc doit être indentée au même niveau.

Concepts de base

Structures conditionnelles

Les structures conditionnelles permettent d'exécuter du code uniquement si certaines conditions sont remplies.

- La structure if
- La structure if … else ….
- La structure if… elif …else
- La structure match

Concepts de base

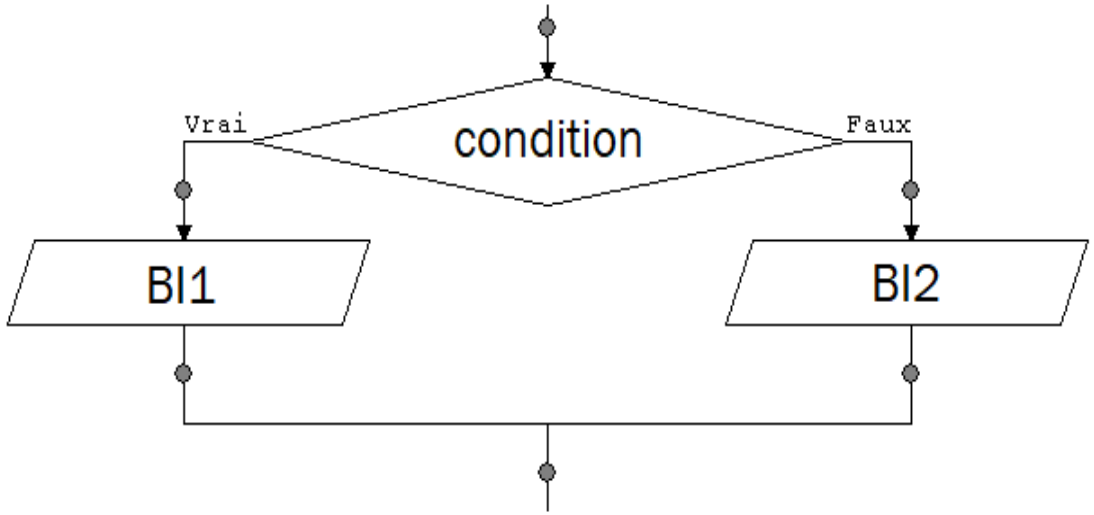
Structures conditionnelles

La structure if … else … :

Syntaxe :

```
if condition :
        BI 1
else :
        BI 2
```

Organigramme:



Concepts de base

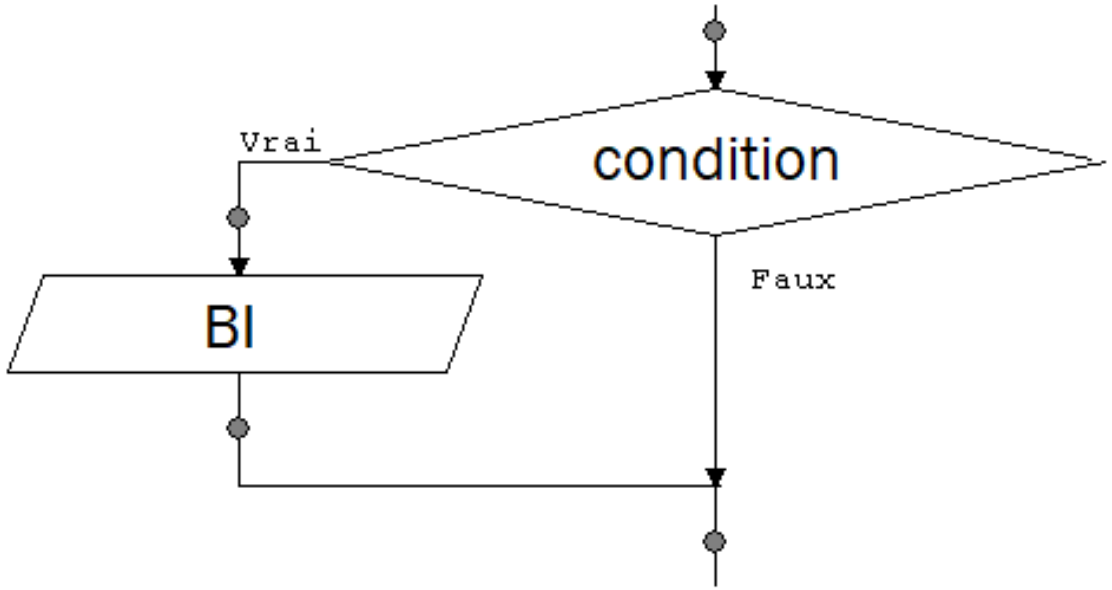
Structures conditionnelles

La structure if … :

Syntaxe :

```
if condition:
        BI
```

Organigramme:



Exemple :

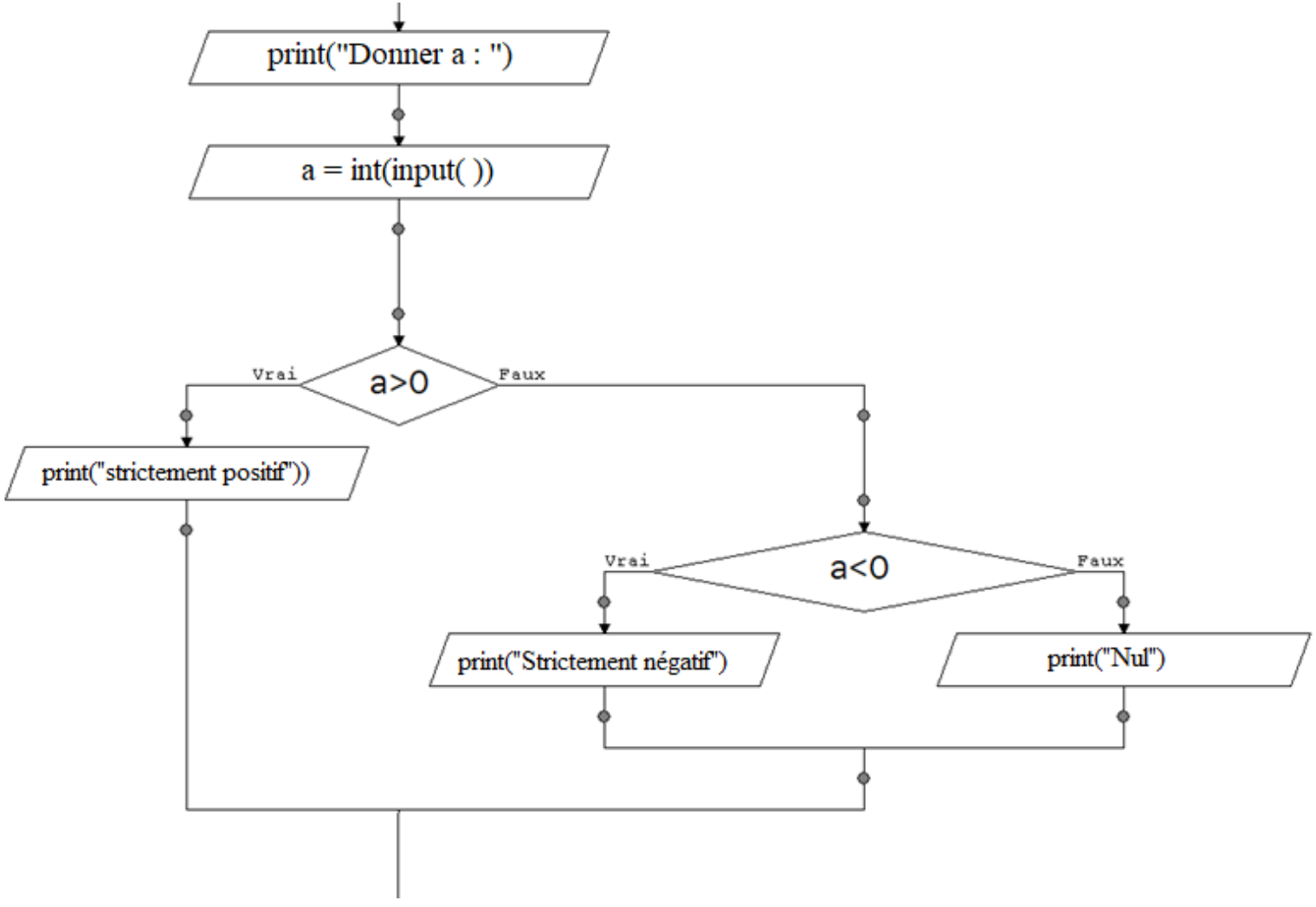
Ecrire un algorithme qui détermine le signe d'un nombre :

- Strictement positif
- Strictement négatif
- nul

Concepts de base

Structures conditionnelles imbriquées

Organigramme :



Structures conditionnelles imbriquées

Solution 1 :

```
print("Donner a : " )
A = int(input())
if a>0 :
      print("Strictement Positif")
else :
      if a<0 :
            print("Strictement Négatif")
      else :
            print("Nul")
```

Structures conditionnelles imbriquées

La structure if … else … :

```
if condition 1 :
        TRAITEMENT 1
elif condition 2 :
        TRAITEMENT 2
elif condition 3:
        TRAITEMENT 3
...
else :
        TRAITEMENT PAR DEFAULT
```

Structures conditionnelles imbriquées

Solution 2 :

```
print("Donner a : " )
A = int(input())


if a>0 :
      print("Strictement Positif")
elif a<0 :
      print("Strictement Négatif")
else:
      print("Nul")
```

Structures conditionnelles imbriquées

La structure match:

```
match variable :
    case valeur1:
        traitement 1
    case valeur1:
        traitement 1
    …
    case _ :
        traitement par défaut
```

Points à retenir :

- match compare des motifs (patterns) et non seulement des valeurs.
- Chaque case peut inclure des motifs spécifiques comme des tuples, des listes, ou des dictionnaires.
- L'underscore (_) est utilisé pour capturer tous les cas non gérés explicitement.

Structures conditionnelles imbriquées

La structure match: Exemple

```
match valeur:
    case (1, x):
        print(f"Tuple avec 1 et {x}")
    case [1, 2, 3]:
        print("Liste exacte [1, 2, 3]")
    case {"clé": y}:
        print(f"Dictionnaire avec 'clé' ayant la valeur {y}")
    case _:
        print("Motif non reconnu")
```

Structures répétitives (Itératives ou Boucles)

Structure indéterministe :

- La boucle : while

Le nombre de répétitions dépend d'une condition

Structure déterministe :

- La boucle: for

Le nombre de répétitions est connu à l'avance

La boucle while

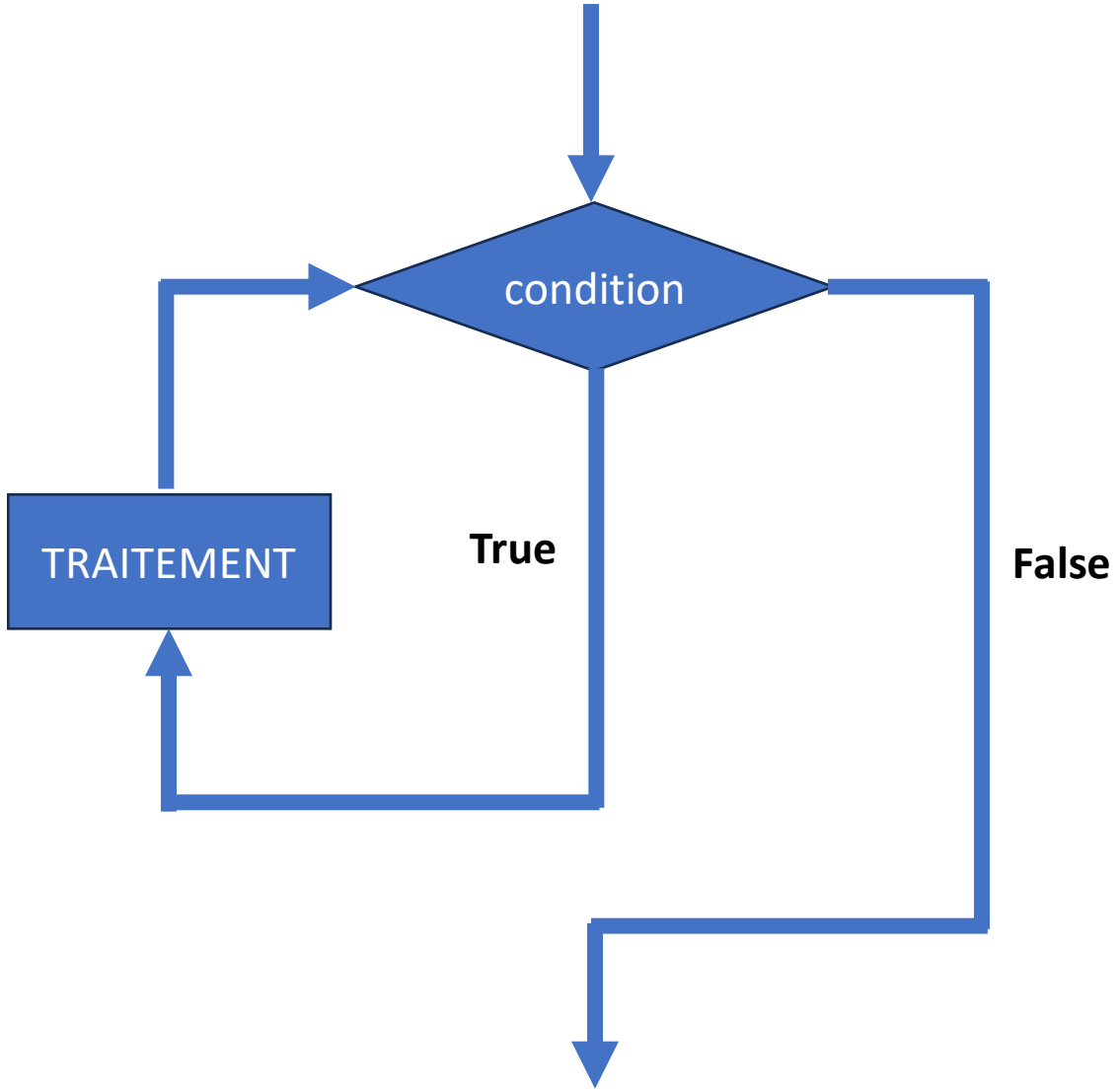
Définition :

La boucle while est une structure de contrôle en programmation qui permet d'exécuter un bloc de code répétitif tant qu'une condition donnée est vraie. Cela signifie que le bloc de code à l'intérieur de la boucle while est exécuté de manière répétée tant que la condition spécifiée reste vraie. Une fois que la condition devient fausse, l'exécution de la boucle s'arrête et le programme passe à l'instruction suivante après la boucle.

La boucle while

Syntaxe :

```
while condition :
    TRAITEMENT
```



La boucle while

Exemple 1 :

```
N=int(input("Donner le nombre de vos enfants : "))
# contrôler la validité des données d'entrés
while N<0:
        N = int(input("Erreur, donner une valeur >= 0 " ))

S = N*300
print("Le montant à verser est :", S)
```

La boucle while

Exemple 2 :

```
R= input("Voulez-vous un café ? O/N : ")
# contrôler la validité des données d'entrés
while R!='O' and R!='N' :
        R = input(« Réessayez .Voulez-vous un café ? O/N : " )

if R=='O' :
        print("Boisson servi !")
else :
        print("Aurevoir!")
```

La boucle while avec else

Syntaxe :

```
while condition :
      TRAITEMENT 1
else:
      TRAITEMENT 2
```

La boucle **while** en Python peut être combinée avec une clause **else** qui s'exécute lorsque la boucle se termine normalement (c'est-à-dire, sans interruption via un break).

La boucle for

Définition :

La boucle for est une structure de contrôle en programmation qui permet d'itérer une séquence de valeurs.

Les séquences en Python :

- Les chaînes de caractères
- Les listes
- Les tuples
- Les clés ou items ou valeurs des dictionnaires

La boucle for

Syntaxe :

```
for i  in séquence :
    TRAITEMENT
```

Fonctionnement :
Pour chaque élément i de la séquence exécuter le TRAIEMENT

La boucle for

Itérer une chaîne de caractères :

```
for i in "Maroc" :
    print(i)
```



```
M
a
r
o
c
```

La boucle for

Itérer les valeurs d'un intervalle [a, b]:

La fonction **range** permet de générer une séquence de nombre entiers :

La syntaxe est :

```
range( début,  fin,  pas)     # Le pas est optionnel et a la valeur 1 par défaut
```

La boucle for

Itérer les valeurs d'un intervalle [a, b]:

Exemples :

range(1 , 6) va générer la séquence 1, 2, 3, 4, 5
range(1 , 11) va générer la séquence 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
range(-2, 3) va générer la séquence -2, -1, 0, 1, 2
range(1 , 11, 2) va générer la séquence 1, 3, 5, 7, 9
range(1 , 11, 3) va générer la séquence 1, 4, 7, 10
range(11) va générer la séquence 0,1, 2, 3, 4, 5, 6, 7, 8, 9, 10
range(11 , 1) va générer la séquence VIDE
range(4 , 1, -1) va générer la séquence 4, 3, 2
range(-4 , -1) va générer -4, -3, -2

La boucle for

Exemple 2:

```
for  i in range(1, 5):
     print("Hello World")
```

```
Hello World
Hello World
Hello World
Hello World
```

La boucle for

Une boucle pour compter :

```
i= 1
while  i<=40 :
        TRAITEMENT
        i = i + 1
```

```
for  i in range(1, 41):
        TRAITEMENT
```

La boucle for avec else

Syntaxe :

```
for  i in sequence:
    TRAITEMENT 1
else:
    TRAITEMENT 2
```

La clause **else** s'exécute après la boucle for si la boucle se termine normalement, sans être interrompue par un break.

Boucles imbriquées

Définition:

```
for  i in séquence:
    Block Instructions
```

```
while  condition:
    Block Instructions
```

Si les instructions d'une boucle contiennent une autre boucle on parle d'une boucle imbriquée.

Boucles imbriquées

Exemples :

```
for  i in range(1, 20):
        for j in range(1, 30):
                Bloc Instructions

while condition:
        for  i in range(1, 20):
                Bloc Instructions
```

```
for  i in range(1, 20):
        while condition:
                Bloc Instructions

while condition1:
        while condition2:
                Bloc Instructions
```

Boucles imbriquées

Fonctionnement :

Pour chaque itération de la première boucle, la boucle imbriquée va s'exécuter jusqu'au bout.

```
for  i in range(1, 4):
        print("i=",i)
        for j in range(1, 3):
                print("   j=",j)
```

```
i=1
   j=1
   j=2
i=2
   j=1
   j=2
i=3
   j=1
   j=2
```

Boucles imbriquées

Exemple:

Afficher les couples de valeurs (x, y) tel que $x \in [1,5]$ et $y \in [1,3]$

```
for  x in range(1, 6) :
    for y in range(1, 4):
        print(x, y)
```

Les instructions break et continue :

Les instructions break et continue sont deux outils essentiels en Python pour contrôler le flux d'exécution à l'intérieur des boucles (for ou while). Elles permettent de modifier le comportement d'une boucle en fonction de conditions spécifiques.

Fonctionnement :

- L'instruction **break** est utilisée pour arrêter immédiatement une boucle.
- L'instruction **continue** permet de passer directement à l'itération suivante de la boucle, sautant le reste du code pour l'itération en cours.

Les instructions break et continue :

Exemple :

```
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
```

```
1
3
5
7
9
```

Boucles imbriquées

Exemple 2:

Comment sortir d'une boucle quand une condition est réalisée dans la boucle imbriquée.

```
for  x in sequence1 :
    for y in sequence2:
        if condition :
            break
```

Ça marche pas, le break utilisé dans la condition ne permet que de sortir de la deuxième boucle et donc continuer vers l'itération suivante de la boucle mère.

Boucles imbriquées

Exemple 2:

Comment sortir d'une boucle quand une condition est réalisée dans la boucle imbriquée.

Solution 1:

```
sortir = False
for  x in sequence1 :
    for y in sequence2:
        if condition :
            sortir = True
            break
    if sortir : break
```

Boucles imbriquées

Exemple 2:

Comment sortir d'une boucle quand une condition est réalisée dans la boucle imbriquée.

Solution 2:

```
for  x in sequence1 :
    for y in sequence2:
        if condition :
            break
    else:
        continue
    break
```

Les fonctions

Les fonctions

Plan

- Définition
- Passage d’arguments aux fonctions
- Retourner une valeur depuis une fonction
- Valeur par défaut d’un argument
- Arguments à longueur variable
- Portée des variables
- La fonction lambda

Introduction

Exemple 1: Ecrire un programme qui calcule C_n^p

```
n = int(input("Donner n : "))
p = int(input("Donner p : "))
# calculer n!
fn = 1
for i in range(2, n+1):
    fn = fn * i
# calculer p!
fp = 1
for i in range(2, p+1):
    fp = fp * i
# calculer (n-p)!
fnp = 1
for i in range(2, n-p+1):
    fnp = fnp * i

# calculer la combinaison
C = fn // (fp*fnp)
print(C)
```

Les fonctions

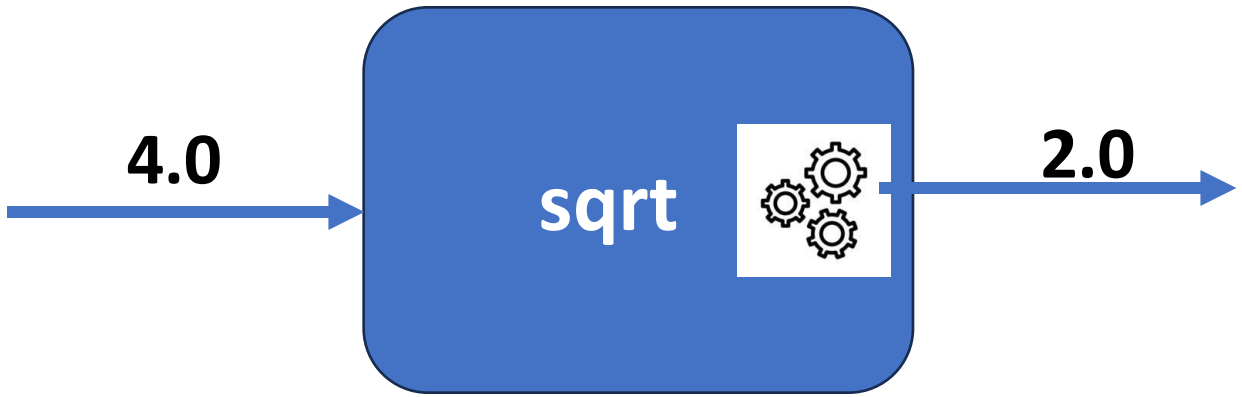
Définition

Une fonction est un bloc de code réutilisable qui effectue une tâche spécifique. Elle peut prendre des paramètres (ou arguments) en entrée, effectuer des opérations sur ces données et renvoyer un résultat.



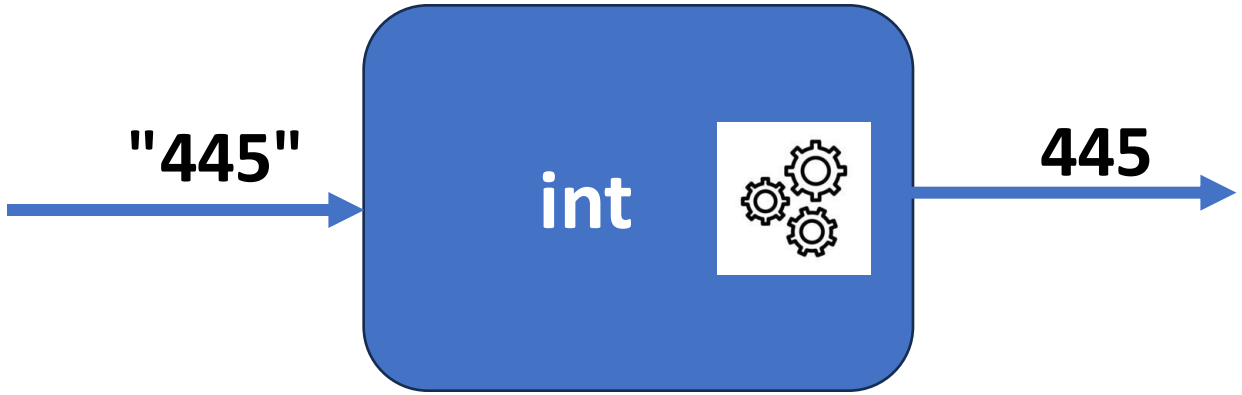
Définition

Fonctions intégrées : **sqrt**



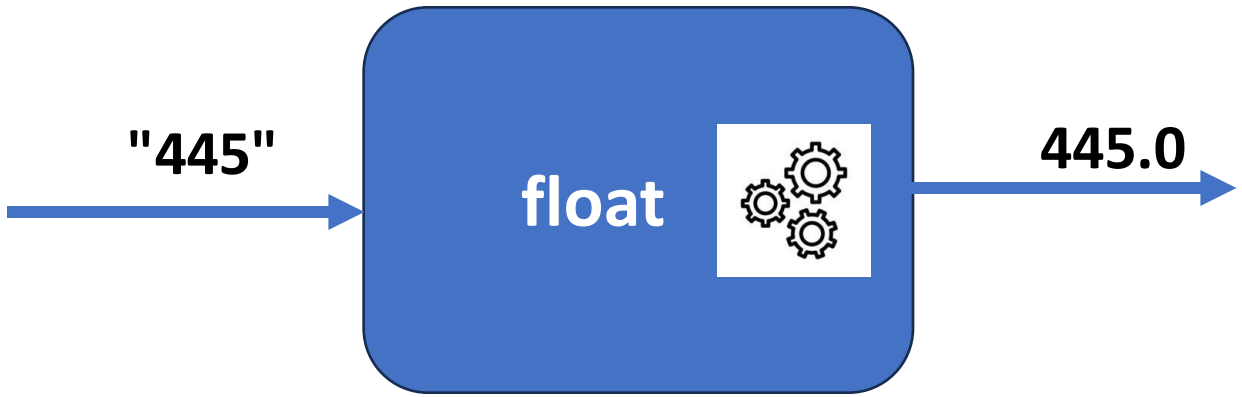
Définition

Fonctions intégrées : **int**



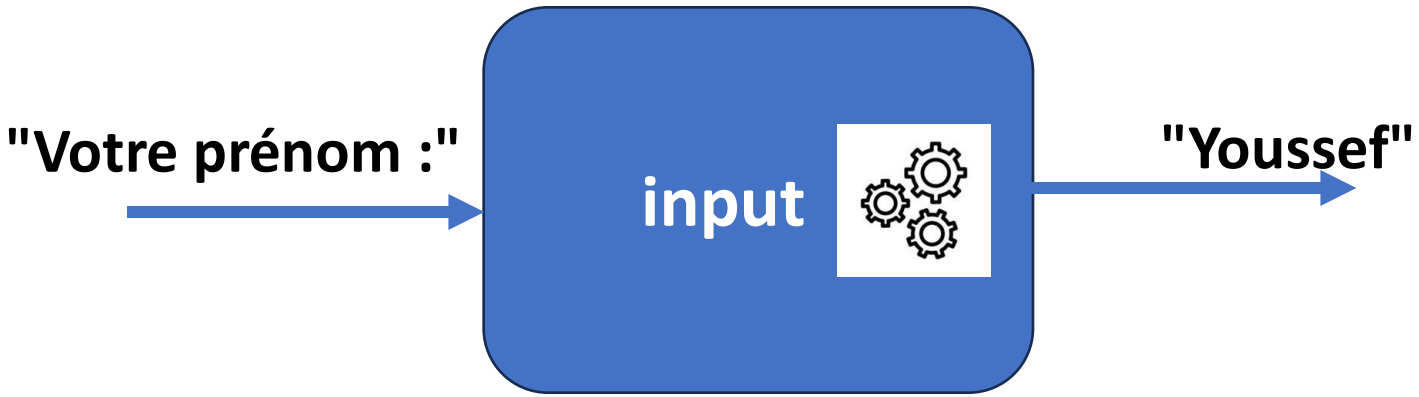
Définition

Fonctions intégrées : float



Définition

Fonctions intégrées : **input**



Définition

Une boîte noir :

Dans le contexte de la programmation, une fonction peut être considérée comme une boîte noire si :

- **Entrées** : Tu fournis des arguments à la fonction (entrées).
- **Sorties** : La fonction renvoie un résultat (sortie).
- **Implémentation cachée** : Tu n'as pas besoin de comprendre comment la fonction réalise ses calculs internes pour l'utiliser. Tu te concentres uniquement sur ce qu'elle fait et comment l'utiliser.

Les fonctions

Syntaxe

En Python, une fonction est un bloc de code réutilisable qui effectue une tâche spécifique. Elle peut prendre des paramètres (ou arguments) en entrée, effectuer des opérations sur ces données et renvoyer un résultat. Voici comment définir une fonction :

```
def nomFonction( argument1, argument2, …) :

    instructions

    [ return résultat ]
```

Les fonctions

Syntaxe

Une fonction qui retourne le max de deux nombres :




```
def nomFonction( argument1, argument2, …) :

        instructions

        [ return résultat ]
```

```
def max( a, b) :
        if a>b :
                mx = a
        else :
                mx = b
        return mx
```

Les fonctions

Syntaxe

Une fonction qui retourne le max de deux nombres :




```
def nomFonction( argument1, argument2, …) :

        instructions

        [ return résultat ]
```

```
def max( a, b) :
    if a>b :
        return a
    else :
        return b
```

Les fonctions

Syntaxe

Une fonction qui retourne le max de deux nombres :




```
def nomFonction( argument1, argument2, …) :

        instructions

        [ return résultat ]
```

```
def max( a, b) :
    if a>b :
        return a
    return b
```

Appel d'une fonction

Une fonction peut être appelé par un programme ou une autre fonction, il suffit d'écrire le nom de la fonction et donner une valeur à chaque argument de la fonction.

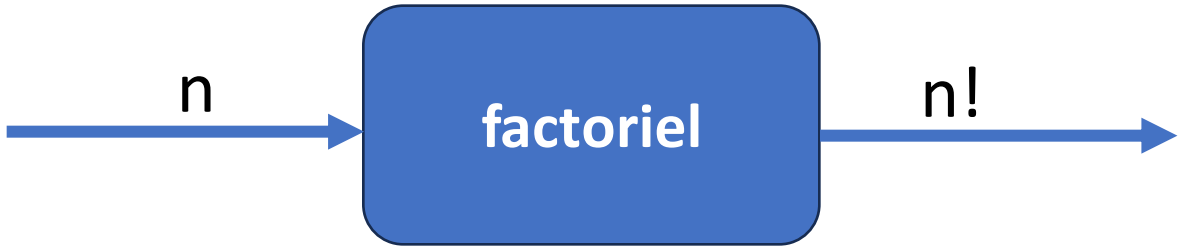
Syntaxe : **variable = nomFonction(Valeur1, Valeur2, …)**

Exemples :

```
x = max(10, 79)
y = max(39, 5)
```

Avantages

Exemple 1: Ecrire un programme qui calcule C_n^p




```
def factoriel(n):
    f = 1
    for i in range(2, n+1):
        f = f * i
    return f
```

Avantages

Exemple 1: Ecrire un programme qui calcule C_n^p

```
n = int(input("Donner n : "))
p = int(input("Donner p : "))
# calculer n!

fn = factoriel(n)

# calculer p!

fp = factoriel(p)

# calculer (n-p)!

fnp = factoriel(n-p)

# calculer la combinaison
C = fn // (fp*fnp)
print(C)
```

Avantages

Exemple 1: Ecrire un programme qui calcule C_n^p

```
n = int(input("Donner n : "))
p = int(input("Donner p : "))

fn = factoriel(n)
fp = factoriel(p)
fnp = factoriel(n-p)

# calculer la combinaison
C = fn // (fp*fnp)
print(C)
```

Avantages

Exemple 1: Ecrire un programme qui calcule C_n^p

```
def factoriel(n):
    f = 1
    for i in range(1, n+1):
        f = f * i
    return f

def combinaison(n, p):
    fn = factoriel(n)
    fp = factoriel(p)
    fnp = factoriel(n-p)
    return fn // (fp*fnp)
```

```
n = int(input("Donner n : "))
p = int(input("Donner p : "))

C = combinaison(n, p)

print(C)
```

Arguments optionnels

Définition :

Python permet aux arguments de fonction d'avoir une valeur par défaut, si la fonction est appelée sans l'argument il a la valeur par défaut. De plus, les arguments peuvent être donnés dans n'importe quel ordre en utilisant les arguments nommés.

Arguments optionnels

Syntaxe :

```
def nomFonction( arg1, arg2, arg3=valeur1, arg4=valeur2) :
    instructions
    [ return résultat ]
```

Arguments optionnels

Exemple 2:

```
def montantTTC(puht , qte, tva) :
      mht = puht * qte
      mtva = mht * tva
      mttc = mht + mtva
      return mttc
```

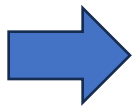
```
M1 = montantTTC(5000 , 10,  0.2)
M2 = montantTTC(7000 , 5  ,  0.2)
M3 = montantTTC(3000 , 10,  0.2)
M4 = montantTTC(100 , 10  ,  0.07)
```

Arguments optionnels

Exemple 2:

```
def montantTTC(puht , qte, tva=0.2) :
    mht = puht * qte
    mtva = mht * tva
    mttc = mht + mtva
    return mttc
```

M1 = montantTTC(5000 , 10, 0.2)
M2 = montantTTC(7000 , 5 , 0.2)
M3 = montantTTC(3000 , 10, 0.2)
M4 = montantTTC(100 , 10 , 0.07)



M1 = montantTTC(5000 , 10)
M2 = montantTTC(7000 , 5)
M3 = montantTTC(3000 , 10)
M4 = montantTTC(100 , 10 , 0.07)

Arguments optionnels

Exemple 2:

```
def montantTTC(puht , qte=10, tva=0.2) :
    mht = puht * qte
    mtva = mht * tva
    mttc = mht + mtva
    return mttc
```

M1 = montantTTC(5000)
M2 = montantTTC(7000 , 5)
~~M4 = montantTTC(100 , 0.07)~~
M4 = montantTTC(100 , tva = 0.07)
M5 = montantTTC(100 , tva = 0.07, qte=15)

Arguments de longueur variable

Définition :

Python permet aux fonctions d'avoir un nombre d'arguments avec une longueur indéfinie.

Arguments de longueur variable

Exemple 3:

```
def max(a , b) :
      if a>b :
            return a
      else:
            return b
```

```
def max(*a) :
      mx = a[0]
      for i in range(1, len(a)):
            if  a[i] > mx :
                  mx = a[i]
      return mx
```

Exercices

Exercice : Donner la signature de la fonction print

print(*messages, end='\n', sep=' ') → None

Exercice : Donner la signature de la fonction max

max(a, b, *args)

Exercice : Donner la signature de la fonction range

range(start, stop=None, step=1)

Les fonctions lambda

Les fonctions lambda en Python sont des fonctions anonymes, c'est-à-dire des fonctions sans nom, qui peuvent avoir un nombre quelconque d'arguments mais ne peuvent contenir qu'une seule expression. Elles sont souvent utilisées pour des opérations simples et courtes.

La syntaxe : lambda arguments : expression

Les fonctions lambda

Exemples:

```
def f(x):
    return x**2
```

Equivalent à :

```
f = lambda x : x**2
```

```
def g(x, y):
    return x+y
```

Equivalent à :

```
g = lambda x,y : x+y
```

```
def h(x):
    if x!=0 :
        return 1/x
    return x
```

Equivalent à :

```
h = lambda x : x!=0 if 1/x else x
```

Porté de variables

La « portée » d'une variable désigne l'espace du script dans laquelle elle va être accessible et visible.

On distingue deux types de portée de variables :

1. **Variable locale**
2. **Variable globale**

Porté de variables

1. Variable locale:

Est une variable déclarée à l'intérieur d'une fonction. Elle n'est accessible qu'à l'intérieur de la fonction dans laquelle elle a été déclaré.

```
def ma_fonction():
        x = 4
        print('ma variable vaut ', x)

ma_fonction()

print(x)
```

Erreur : x est non définie

Porté de variables

2. Variable globale:

Est une variable déclarée en dehors des fonctions. Elle visible et utilisable dans toute les fonctions (**mais la fonction ne peut le modifier**).

```
total = 10
def ma_fonction():
    x = total + 4
    print('ma variable vaut ', x)

ma_fonction()
print(total)
```

Porté de variables

2. Variable globale:

Est une variable déclarée en dehors des fonctions. Elle visible et utilisable dans toute les fonctions (**mais la fonction ne peut le modifier**).

```
total = 10
def ma_fonction():
    total = 100
    print('ma variable vaut ‘, total)

ma_fonction()
print(total)
```

```
total = 10
def ma_fonction():
    global total
    total = 100
    print('ma variable vaut ‘, total)

ma_fonction()
print(total)
```

Porté de variables

Résumé de l'ordre de recherche:

L'ordre de recherche des variables en Python est donc :

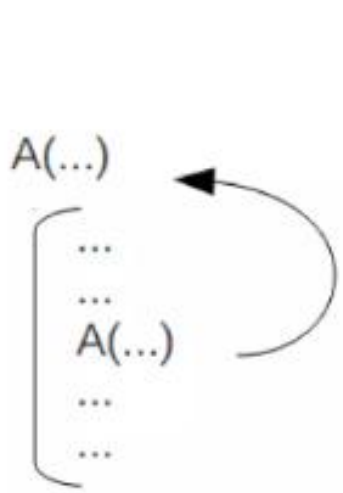
1. **Locale** : Variables définies dans la fonction actuelle.
2. **Englobante** : Variables définies dans la fonction englobante (si la fonction est imbriquée).
3. **Globale** : Variables définies au niveau du module.

Les fonctions récursives

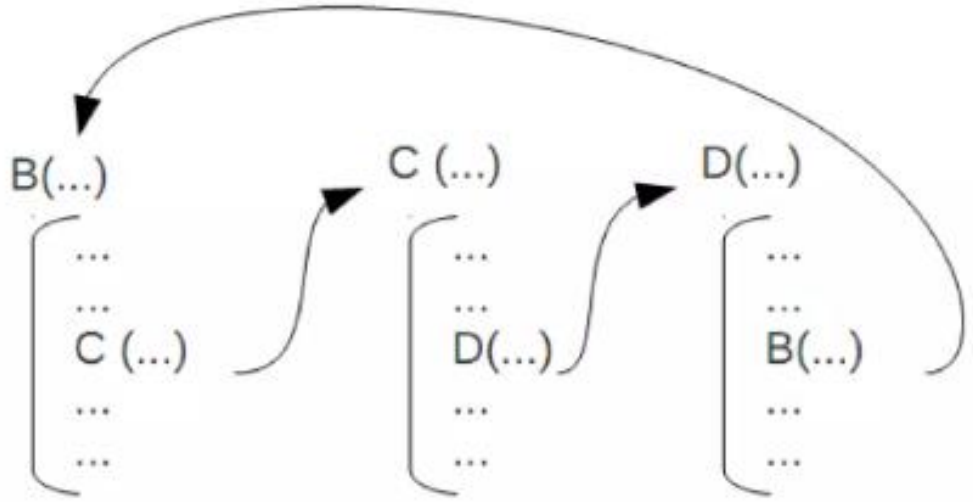
Les fonctions récursives

Définition

- Une fonction est dite récursive lorsqu'elle est auto référente, c-à-d elle fait référence à elle-même dans sa définition.
- Autrement dit, elle est définit en fonction d'elle-même, directement ou indirectement.



Récursivité directe



Récursivité indirecte ou mutuelle

Les fonctions récursives

Exemple

- Calculer n!

```
def factoriel(n):
    f = 1
    for i in range(2, n+1):
        f = f * i
    return f
```

Version Itérative

Les fonctions récursives

Exemple

- Calculer n!

```
n! =  n*(n-1)*(n-2)*…*1
n! =  n*(n-1)!

factoriel(n) = n* factoriel(n-1)
```

Les fonctions récursives

Exemple

- Calculer n!

```
def factoriel(n):
    return n*factoriel(n-1)
```


Appel récursif

Version Récursive

Evolution d'un appel récursif

L'exécution d'un appel récursif passe par deux phases, la phase de descente et la phase de la remontée :

- Dans la phase de descente, chaque appel récursif fait à son tour un appel récursif.

```
def factoriel(n):
    return n*factoriel(n-1)
```

factoriel(4) ← 4 * factoriel(3)

factoriel(3) ← 3 * factoriel(2)

factoriel(2) ← 2 * factoriel(1)

Phase de descente

factoriel(1) ← 1 * factoriel(0)

factoriel(0) ← 0 * factoriel(-1)

Evolution d'un appel récursif

Puisqu'un algorithme récursif s'appelle lui-même, il est impératif qu'on prévoit une condition d'arrêt à la récursion, sinon le programme ne s'arrête jamais!

```
def factoriel(n):
    if n==0 :
        return 1
    return n*factoriel(n-1)
```

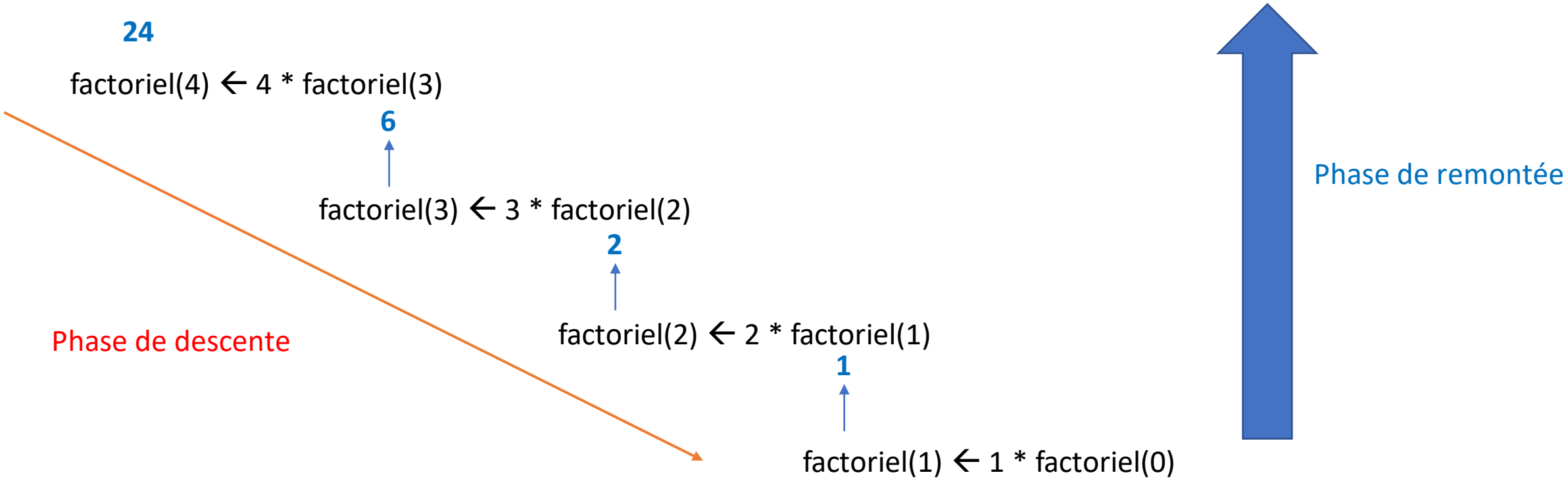
Condition d'arrêt

Appel récursif

Evolution d'un appel récursif

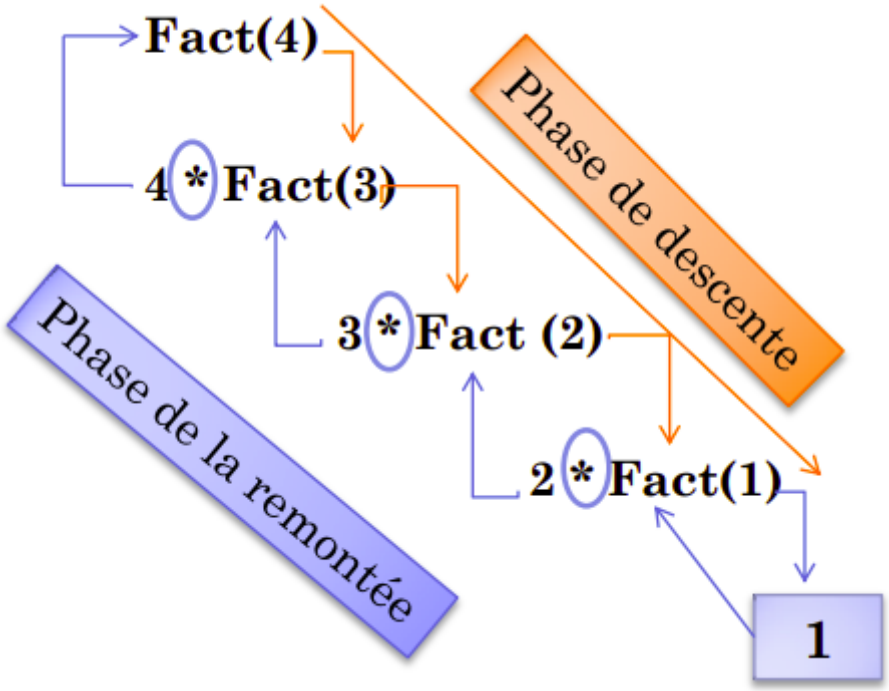
L'exécution d'un appel récursif passe par deux phases, la phase de descente et la phase de la remontée :

- En arrivant à la condition terminale, on commence la phase de la remontée qui se poursuit jusqu'à ce que l'appel initial soit terminé, ce qui termine le processus récursif.



Récursivité Terminale est Non terminale

- Une fonction récursive est dite non terminale si le résultat de l'appel récursif est utilisé pour réaliser un traitement (en plus du retour de la fonction).




```
def Fact(n):
    if n==0 :
        return 1
    return n*Fact(n-1)
```

Récursivité Terminale est Non terminale

- PARFOIS, il est possible de transformer une fonction récursive non terminale à une fonction récursive terminale.

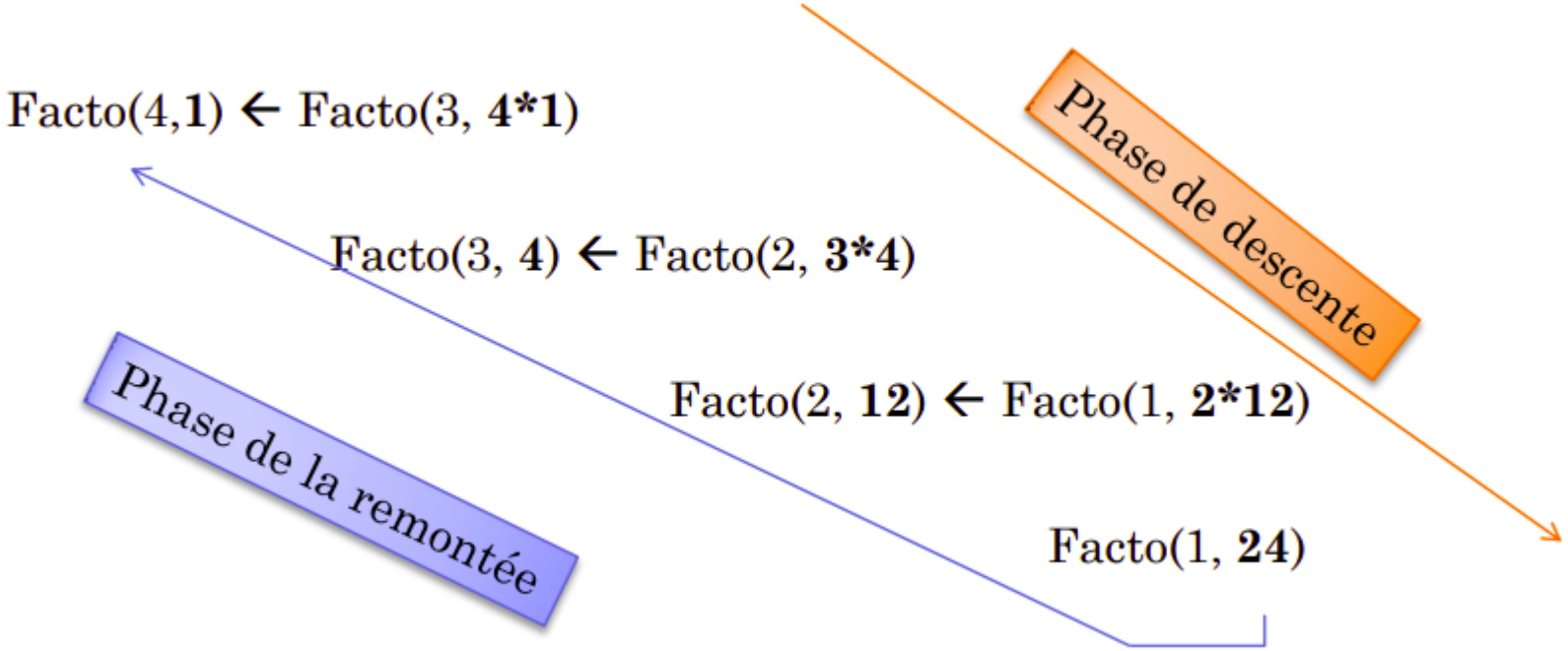
```
def factoriel(n):
    if n==0 :
        return 1
    return n*factoriel(n-1)
```

```
def factoriel(n, resultat):
    if n==1 :
        return resultat
    return factoriel(n-1,n*resultat)
```

NB : la fonction doit être appelée en mettant resultat à 1

Récursivité Terminale est Non terminale

- PARFOIS, il est possible de transformer une fonction récursive non terminale à une fonction récursive terminale.



FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

DÉPARTEMENT DE MATHÉMATIQUES ET INFORMATIQUE

Les structures de données (Les Tuples, les listes, les dictionnaires, les ensembles)

Master : MNSA

Pr: Youssef Ouassit

Structures de données

Définition :

Les structures de données sont des moyens d'organiser et de stocker des données afin qu'elles puissent être utilisées de manière efficace. En Python, plusieurs structures de données sont disponibles, chacune adaptée à différents types de tâches. Voici une présentation des principales structures de données en Python.

Structures de données

Plusieurs catégories :

Structures linéaires:

- Str
- Liste
- Tuple
- Sets
- Dicts

Structures non linéaires:

- Arbres
- Graphes

Structures de donnée

Définition :

Les structures de données essentielles:

Standards	Importées par des modules
list tuple str set dict	Stack Queue Linked list Tree …

Les listes

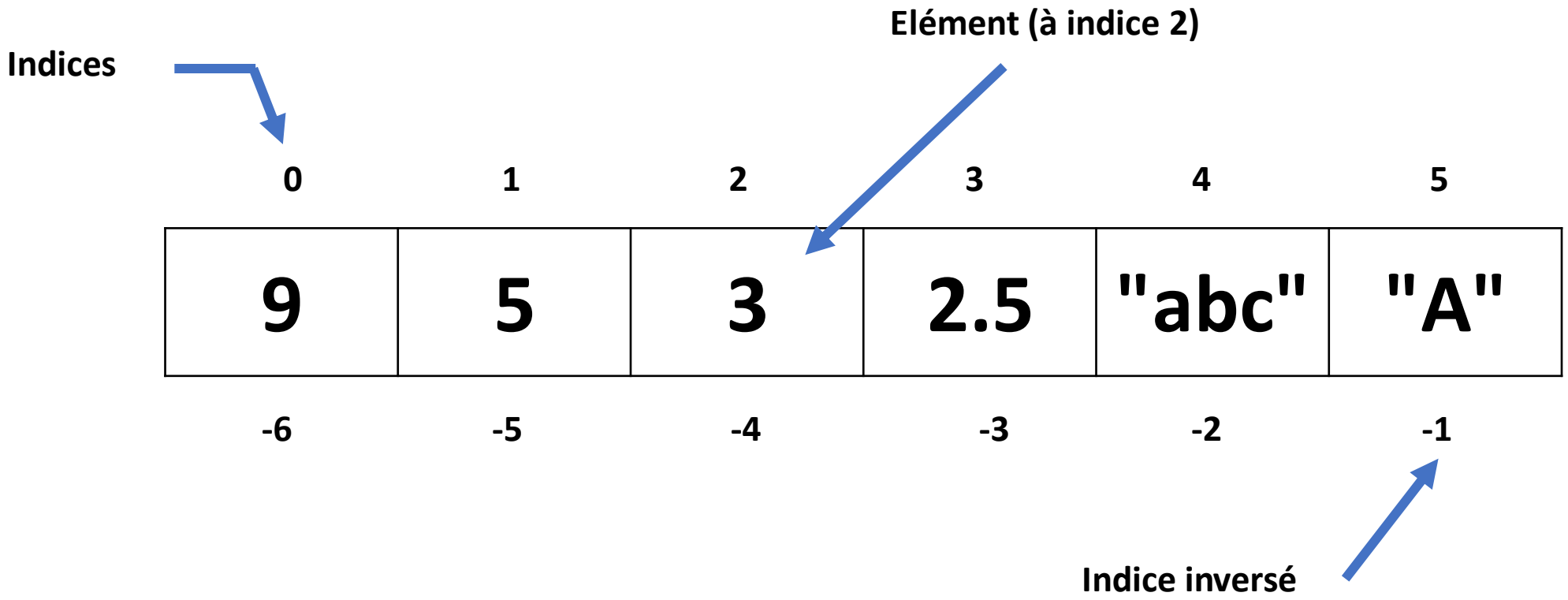
Définition d’une liste

Une liste est une structure de données flexible et mutable qui peut contenir un ensemble ordonné d'éléments de différents types.

1. Une liste une collection de données ordonnée.
2. Permet de manipuler une séquence d’objets modifiables.
3. Contient un nombre non fixé d’éléments.
4. Contient des données de différents types.

Définition d'une liste

Une liste est une séquence d'éléments linéaire, que nous pouvons représenter graphiquement sous la forme suivante :



Les listes

Déclaration

```
# Liste vide
ma_liste_vide = []

# Liste avec des éléments
ma_liste = [1, 2, 3, 4, 5]

# Liste avec des types différents
ma_liste_melangee = [1, "Python", 3.14, True]
```

Les listes

Déclaration

```
# Liste vide avec list()
ma_liste_vide = list()

# Liste à partir d'une chaîne de caractères
ma_liste_chaine = list("Python") # ['P','y','t','h','o','n']

# Liste à partir d'un tuple
ma_liste_tuple = list((1, 2, 3))  # [1, 2, 3]
```

Déclaration

La "list comprehension" :

```
# Liste des carrés des nombres de 0 à 9
carres = [x**2 for x in range(10)]
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# Liste des nombres pairs de 0 à 9
pairs = [x for x in range(10) if x % 2 == 0]
# [0, 2, 4, 6, 8]
```

Opérateurs de base

Opérateur d'Appartenance **in**

```
fruits = ["pomme", "banane", "cerise"]
print("pomme" in fruits)  # True
print("orange" in fruits)  # False
```

Opérateur de concaténation **+**

```
liste1 = [1, 2, 3]
liste2 = [4, 5, 6]
liste3 = liste1 + liste2  # [1, 2, 3, 4, 5, 6]
```

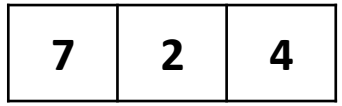
Les listes

- Opérateurs
- Fonctions
- Méthodes

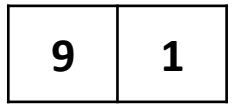
Opérateurs de base

Opérateur de concaténation :

A = [7 , 2 , 4]



B = [9 , 1]



C = A + B

Résultat :

C = [7 , 2 , 4 , 9 , 1]



Opérateurs de base

Opérateur de répétition *

```
liste = [1, 2, 3]
resultat = liste * 3  # [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Opérateur d’indexation []

```
fruits = ["pomme", "banane", "cerise"]
print(fruits[0])  # 'pomme'
print(fruits[-1])  # 'cerise'
```

Opérateurs de base

Opérateur de Slicing **[debut : fin : pas]**

```
chiffres = [0, 1, 2, 3, 4, 5]
sous_liste = chiffres[1:4]  # [1, 2, 3]
sous_liste2 = chiffres[:3]   # [0, 1, 2]
sous_liste3 = chiffres[::2]  # [0, 2, 4]
sous_liste3 = chiffres[:]  # [0, 1, 2, 3, 4, 5]
```

Les fonctions

Nombre des éléments:

La fonction **len** permet de calculer le nombre des éléments d'une liste:

```
L = [ 5  ,  8 ,  1 ]

n = len(L)
n = 3
```

Valeur maximale d'une liste:

La fonction **max** permet de calculer le maximum des éléments d'une liste:

```
L = [ 5  ,  8 ,  1 ]

n = max(L)
n = 8
```

Valeur minimale d'une liste:

La fonction **min** permet de calculer le minimum des éléments d'une liste:

```
L = [ 5  ,  8 ,  1 ]

n = min(L)
n = 1
```

Somme des éléments d'une liste:

La fonction **sum** permet de calculer la somme des éléments d'une liste:

```
L = [ 5  ,  8 ,  1 ]

n = sum(L)
n = 14
```

Trier les éléments d'une liste:

La fonction **sorted** permet de trier les éléments d'une liste:

```
L = [ 5  ,  8 ,  1 ]

A = sorted(L)
A = [1, 5, 8]
B = sorted(L, reverse=True)
B = [8, 5, 1]
```

Les listes

Les méthodes

Méthode `append()`

Description : Ajoute un élément à la fin de la liste.

Syntaxe : `liste.append(élément)`

Exemple :

```python
fruits = ["pomme", "banane"]
fruits.append("cerise")  # ["pomme", "banane", "cerise"]
```

Les listes

Les méthodes

Méthode `extend()`

Description : Ajoute les éléments d'une autre liste à la fin de la liste courante.

Syntaxe : `liste.extend(autre_liste)`

Exemple :

python

```python
liste1 = [1, 2, 3]
liste2 = [4, 5, 6]
liste1.extend(liste2)  # [1, 2, 3, 4, 5, 6]
```

Les listes

Les méthodes

Méthode `insert()`

Description : Insère un élément à une position spécifique de la liste.

Syntaxe : `liste.insert(index, élément)`

Exemple :

```python
fruits = ["pomme", "cerise"]
fruits.insert(1, "banane")  # ["pomme", "banane", "cerise"]
```

Les listes

Les méthodes

Méthode `remove()`

Description : Supprime la première occurrence d'un élément dans la liste.

Syntaxe : `liste.remove(élément)`

Exemple :

```python
fruits = ["pomme", "banane", "cerise", "banane"]
fruits.remove("banane")  # ["pomme", "cerise", "banane"]
```

Les listes

Les méthodes

Méthode `pop()`

Description : Supprime et retourne l'élément à une position donnée (par défaut, le dernier).

Syntaxe : `liste.pop([index])`

Exemple :

```python
fruits = ["pomme", "banane", "cerise"]
dernier_fruit = fruits.pop()  # "cerise"
```

Les listes

Les méthodes

Méthode `clear()`

Description : Supprime tous les éléments de la liste.

Syntaxe : `liste.clear()`

Exemple :

```python
fruits = ["pomme", "banane", "cerise"]
fruits.clear()  # []
```

Les listes

Les méthodes

Méthode `index()`

Description : Retourne l'index de la première occurrence d'un élément dans la liste.

Syntaxe : `liste.index(élément)`

Exemple :

python

```python
fruits = ["pomme", "banane", "cerise"]
index_banane = fruits.index("banane")  # 1
```

Les listes

Les méthodes

Méthode `count()`

Description : Compte le nombre de fois qu'un élément apparaît dans la liste.

Syntaxe : `liste.count(élément)`

Exemple :

python

```python
fruits = ["pomme", "banane", "cerise", "banane"]
nombre_banane = fruits.count("banane")  # 2
```

Les listes

Les méthodes

Méthode `sort()`

Description : Trie les éléments de la liste en place.

Syntaxe : `liste.sort([key=None, reverse=False])`

Exemple :

python Copier le code

```python
chiffres = [3, 1, 4, 1, 5, 9]
chiffres.sort()  # [1, 1, 3, 4, 5, 9]
```

Les listes

Les méthodes

Méthode `reverse()`

Description : Inverse l'ordre des éléments de la liste en place.

Syntaxe : `liste.reverse()`

Exemple :

```python
chiffres = [1, 2, 3]
chiffres.reverse()  # [3, 2, 1]
```

Les listes

Les méthodes

Méthode `copy()`

Description : Retourne une copie superficielle de la liste.

Syntaxe : `liste.copy()`

Exemple :

python Copier le code

```python
fruits = ["pomme", "banane", "cerise"]
fruits_copie = fruits.copy()  # ["pomme", "banane", "cerise"]
```

Les listes

Les méthodes

Méthode : `clear`

Description : Supprime tous les éléments de la liste.

Syntaxe : `liste.clear()`

Exemple :

```python
liste = [1, 2, 3]
liste.clear()
print(liste)  # Affiche : []
```

Les tuples

Définition d’un tuple

Un tuple est une séquence ordonnée d'éléments, qui peuvent être de types différents (par exemple, entiers, chaînes de caractères, listes, etc.).

Contrairement aux listes, les tuples sont immuables, ce qui signifie que leurs éléments ne peuvent pas être modifiés après leur création. Une fois qu'un tuple est défini, il ne peut pas être modifié, étendu ou réduit.

Les tuples

Déclaration

```
# Tuple vide
tuple_vide = ()

# Tuple avec des éléments
mon_tuple = (1, 2, 3, 4, 5)

# Tuple avec des éléments sans parenthèses
mon_tuple = 1, 2, 3, 4, 5

# Tuple avec des types différents
mon_tuple_melangee = (1, "Python", 3.14, True)
```

Les tuples

Déclaration

```
# Tuple vide avec tuple()
mon_tuple_vide = tuple()

# Tuple à partir d'une chaîne de caractères
mon_tuple_chaine = tuple("Python") # ('P','y','t','h','o','n’)

# Tuple à partir d’une liste
mon_tuple_tuple = tuple([1, 2, 3])  # (1, 2, 3)

# Tuple avec un seul élément
mon_tuple_vide = (1, )
```

Les tuples

Déclaration

```
# Tuple des carrés des nombres de 0 à 9
carres = (x**2 for x in range(10))
# (0, 1, 4, 9, 16, 25, 36, 49, 64, 81)

# Tuple des nombres pairs de 0 à 9
pairs = (x for x in range(10) if x % 2 == 0)
# (0, 2, 4, 6, 8)
```

Opérateurs de base

Opérateur d'Appartenance **in**

```
fruits = ("pomme", "banane", "cerise" )
print("pomme" in fruits)  # True
print("orange" in fruits)  # False
```

Opérateur de concaténation **+**

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
tuple3 = tuple1 + tuple2  # (1, 2, 3, 4, 5, 6)
```

Les tuples

Opérateurs de base

Opérateur de répétition *

```
tuple = [1, 2, 3]
resultat = tuple * 3  # (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Opérateur d’indexation []

```
fruits = ("pomme", "banane", "cerise« )
print(fruits[0])  # 'pomme'
print(fruits[-1])  # 'cerise'
```

Les tuples

Opérateurs de base

Opérateur de Slicing **[debut : fin : pas]**

```
chiffres = (0, 1, 2, 3, 4, 5)
sous_tuple = chiffres[1:4]  # (1, 2, 3)
sous_tuple2 = chiffres[:3]   # (0, 1, 2)
sous_tuple3 = chiffres[::2]  # (0, 2, 4)
sous_tuple4 = chiffres[:]  # (0, 1, 2, 3, 4, 5)
```

Les tuples

Les méthodes

Méthode : `count`

Description : Retourne le nombre d'occurrences d'un élément spécifique dans le tuple.

Syntaxe : `tuple.count(element)`

Exemple :

```python
mon_tuple = (1, 2, 2, 3)
print(mon_tuple.count(2))  # Affiche : 2
```

Les tuples

Les méthodes

Méthode : `index`

Description : Retourne l'index de la première occurrence d'un élément spécifique dans le tuple. Lève une exception `ValueError` si l'élément n'est pas trouvé.

Syntaxe : `tuple.index(element, start=0, end=len(tuple))`

Exemple :

```python
mon_tuple = (1, 2, 3)
print(mon_tuple.index(2))  # Affiche : 1
```

Les tuples

Les fonctions

Les fonctions sur les listes sont aussi applicable sur les tuples :

- Len
- min
- max
- sum
- sorted
- reversed
- zip
- any
- all

Les str

Définition d’une chaîne de caractères

Le type **str** en Python est une séquence de caractères Unicode.
Les chaînes de caractères sont utilisées pour stocker et manipuler du texte.

Les chaînes de caractères sont définies en utilisant des guillemets simples ('), des guillemets doubles ("), ou des triples guillemets (''' ou """) pour les chaînes multilignes.

Les chaînes de caractères

Déclaration

```
# Définir une chaîne

chaine1 = 'Bonjour'

chaine2 = "le monde"

chaine3 = '''Ceci est une chaîne qui s'étend sur
plusieurs lignes.'''
```

Les chaînes de caractères

Opérateurs de base

Opérateur d'Appartenance **in**

```
ch = "pomme"
print("po" in ch)  # True
print("ora" in ch)  # False
```

Opérateur de concaténation **+**

```
ch1 = "Bonjour "
ch2 = "Ahmed"
ch3 = ch1 + ch2  # "Bonjour Ahmed"
```

Les chaînes de caractères

Opérateurs de base

Opérateur de répétition *

```
ch = "bon"
resultat = ch * 2  # "bonbon"
```

Opérateur d'indexation []

```
ch = "pomme"
print(ch[0])  # 'p'
print(ch[-1])  # 'e'
```

Les chaînes de caractères

Opérateurs de base

Opérateur de Slicing **[debut : fin : pas]**

```
ch = "Bonjour"
sous_chaine = ch[:3]  # Bon
sous_chaine2 = ch[1:3]   # on
sous_chaine3 = ch[::2]  # Bnor
sous_chaine4 = ch[:]  # Bonjour
```

Les chaînes de caractères

Les méthodes

Méthode : `upper`

Description : Retourne une nouvelle chaîne où tous les caractères sont convertis en majuscules.

Syntaxe : `str.upper()`

Exemple :

```python
texte = "bonjour"
print(texte.upper())  # Affiche : BONJOUR
```

Les chaînes de caractères

Les méthodes

Méthode : `lower`

Description : Retourne une nouvelle chaîne où tous les caractères sont convertis en minuscules.

Syntaxe : `str.lower()`

Exemple :

```python
texte = "BONJOUR"
print(texte.lower())  # Affiche : bonjour
```

Les chaînes de caractères

Les méthodes

Méthode : `strip`

Description : Supprime les espaces (ou autres caractères spécifiés) au début et à la fin de la chaîne.

Syntaxe : `str.strip([caractères])`

Exemple :

```python
texte = "  bonjour  "
print(texte.strip())  # Affiche : bonjour
```

Les chaînes de caractères

Les méthodes

Méthode : `replace`

Description : Remplace toutes les occurrences d'une sous-chaîne par une autre dans la chaîne d'origine.

Syntaxe : `str.replace(old, new[, count])`

Exemple :

```python
texte = "bonjour monde"
print(texte.replace("monde", "tout le monde"))  # Affiche : bonjour tout le monde
```

Les chaînes de caractères

Les méthodes

Méthode : `split`

Description : Divise la chaîne en une liste de sous-chaînes en utilisant le séparateur spécifié. Si aucun séparateur n'est fourni, les espaces sont utilisés par défaut.

Syntaxe : `str.split([separator[, maxsplit]])`

Exemple :

python

```python
texte = "bonjour tout le monde"
print(texte.split())  # Affiche : ['bonjour', 'tout', 'le', 'monde']
```

Les chaînes de caractères

Les méthodes

Méthode : `join`

Description : Concatène les éléments d'un itérable (comme une liste) en une seule chaîne, en utilisant la chaîne sur laquelle la méthode est appelée comme séparateur.

Syntaxe : `str.join(iterable)`

Exemple :

python Copier le code

```python
liste = ["bonjour", "tout", "le", "monde"]
print(" ".join(liste))  # Affiche : bonjour tout le monde
```

Les chaînes de caractères

Les méthodes

Méthode : `` `find` ``

Description : Retourne l'index de la première occurrence de la sous-chaîne spécifiée dans la chaîne. Retourne `` `-1` `` si la sous-chaîne n'est pas trouvée.

Syntaxe : `` `str.find(sub[, start[, end]])` ``

Exemple :

python Copier le code

```python
texte = "bonjour"
print(texte.find("jour"))  # Affiche : 3
```

Les chaînes de caractères

Les méthodes

Méthode : `count`

Description : Retourne le nombre de fois qu'une sous-chaîne apparaît dans la chaîne.

Syntaxe : `str.count(sub[, start[, end]])`

Exemple :

```python
texte = "bonjour tout le monde"
print(texte.count("o"))  # Affiche : 3
```

Les chaînes de caractères

Les fonctions

On retrouve les même fonctions déjà vu :

- len
- max
- min
- …

Des fonctions spécifiques aux chaînes de caractères:

- **ord** : Retourne le code Unicode d'un caractère.
- **chr** : Retourne le caractère correspondant à un code Unicode
- **eval :** Évalue une chaîne comme une expression Python.
- **input** : Demande une entrée utilisateur.

Les dictionnaires

Définition

- Un dictionnaire est une collection non ordonnée de paires clé-valeur.

- Chaque clé dans un dictionnaire doit être unique et immuable (par exemple, une chaîne de caractères, un nombre, un tuple), tandis que les valeurs peuvent être de n'importe quel type (chaîne, nombre, liste, autre dictionnaire, etc.).

Les dictionnaires

Déclaration

```
# Définir un dictionnaire vide avec accolades
D = { }

# Définir un dictionnaire vide avec dict
D = dict()

# Exemple d'un dictionnaire simple
etudiant = {
      "nom": "Alice",
      "âge": 25,
      "ville": "Paris"
}
```

Les dictionnaires

Déclaration

```
# À partir d'une liste de tuples
liste_tuples = [("nom", "Alice"), ("âge", 25), ("ville", "Paris")]
etudiant = dict(liste_tuples)

# À partir de deux listes
clés = ["nom", "âge", "ville"]
valeurs = ["Alice", 25, "Paris"]
etudiant = dict(zip(clés, valeurs))
```

Opérateurs de base

Opérateur d'Appartenance **in**

```
D = {"nom": "Alice", "âge": 25}
print("nom" in D)  # True
print("adresse" in D)  # False
```

Les dictionnaires

Opérateurs de base

Opérateur de l'union |

```
dict1 = {"nom": "Alice", "âge": 25}
dict2 = {"ville": "Paris", "âge": 26}
result = dict1 | dict2
# Résultat : {"nom": "Alice", "âge": 26, "ville": "Paris"}
```

Opérateur d’indexation []

```
D = {"nom": "Alice", "âge": 25}
print(D['nom'])  # 'Alice'
print(D['âge'])  # 25
```

Les dictionnaires

Les méthodes

Méthode `clear()`

Description : Supprime toutes les paires clé-valeur du dictionnaire.

Syntaxe : `dictionnaire.clear()`

Exemple :

```python
mon_dict = {"nom": "Alice", "âge": 25}
mon_dict.clear()
print(mon_dict)  # Affiche : {}
```

Les dictionnaires

Les méthodes

Méthode `copy()`

Description : Retourne une copie superficielle du dictionnaire.

Syntaxe : `dictionnaire.copy()`

Exemple :

```python
mon_dict = {"nom": "Alice", "âge": 25}
copie_dict = mon_dict.copy()
print(copie_dict)  # Affiche : {'nom': 'Alice', 'âge': 25}
```

Les dictionnaires

Les méthodes

Méthode `fromkeys()`

Description : Crée un nouveau dictionnaire avec les clés fournies et la même valeur pour toutes les clés.

Syntaxe : `dict.fromkeys(clés, valeur)`

Exemple :

```python
clés = ["nom", "âge", "ville"]
mon_dict = dict.fromkeys(clés, "Inconnu")
print(mon_dict)  # Affiche : {'nom': 'Inconnu', 'âge': 'Inconnu', 'ville': 'Inconnu'}
```

Les dictionnaires

Les méthodes

Méthode `get()`

Description : Retourne la valeur associée à une clé. Si la clé n'existe pas, retourne une valeur par défaut.

Syntaxe : `dictionnaire.get(clé, valeur_par_défaut)`

Exemple :

python Copier le code

```python
mon_dict = {"nom": "Alice", "âge": 25}
age = mon_dict.get("âge", "Non spécifié")
print(age)  # Affiche : 25
```

Les dictionnaires

Les méthodes

Méthode `items()`

Description : Retourne une vue dynamique contenant les paires clé-valeur du dictionnaire sous forme de tuples.

Syntaxe : `dictionnaire.items()`

Exemple :

python Copier le code

```python
mon_dict = {"nom": "Alice", "âge": 25}
print(mon_dict.items())  # Affiche : dict_items([('nom', 'Alice'), ('âge', 25)])
```

Les dictionnaires

Les méthodes

Méthode `keys()`

Description : Retourne une vue dynamique contenant les clés du dictionnaire.

Syntaxe : `dictionnaire.keys()`

Exemple :

```python
mon_dict = {"nom": "Alice", "âge": 25}
print(mon_dict.keys())  # Affiche : dict_keys(['nom', 'âge'])
```

Les dictionnaires

Les méthodes

Méthode `pop()`

Description : Supprime et retourne la valeur associée à une clé. Si la clé n'existe pas, lève une exception `KeyError`, à moins qu'une valeur par défaut ne soit spécifiée.

Syntaxe : `dictionnaire.pop(clé, valeur_par_défaut)`

Exemple :

python

```python
mon_dict = {"nom": "Alice", "âge": 25}
age = mon_dict.pop("âge")
print(age)  # Affiche : 25
print(mon_dict)  # Affiche : {'nom': 'Alice'}
```

Les dictionnaires

Les méthodes

Méthode `popitem()`

Description : Supprime et retourne la dernière paire clé-valeur insérée dans le dictionnaire sous forme de tuple. Si le dictionnaire est vide, lève une exception `KeyError`.

Syntaxe : `dictionnaire.popitem()`

Exemple :

```python
mon_dict = {"nom": "Alice", "âge": 25}
item = mon_dict.popitem()
print(item)  # Affiche : ('âge', 25)
print(mon_dict)  # Affiche : {'nom': 'Alice'}
```

Les dictionnaires

Les méthodes

Méthode `values()`

Description : Retourne une vue dynamique contenant les valeurs du dictionnaire.

Syntaxe : `dictionnaire.values()`

Exemple :

```python
mon_dict = {"nom": "Alice", "âge": 25}
print(mon_dict.values())  # Affiche : dict_values(['Alice', 25])
```

Les dictionnaires

Les méthodes

Méthode `update()`

Description : Met à jour le dictionnaire avec les paires clé-valeur d'un autre dictionnaire ou d'une séquence de paires clé-valeur.

Syntaxe : `dictionnaire.update(autre_dictionnaire)`

Exemple :

python

```python
mon_dict = {"nom": "Alice", "âge": 25}
mise_a_jour = {"ville": "Paris", "âge": 26}
mon_dict.update(mise_a_jour)
print(mon_dict)  # Affiche : {'nom': 'Alice', 'âge': 26, 'ville': 'Paris'}
```

Les ensembles

Définition

En Python, un set (ensemble) est une collection non ordonnée d'éléments uniques. Les ensembles sont utiles pour stocker des valeurs uniques et effectuer des opérations ensemblistes comme l'union, l'intersection et la différence.

Les ensembles

Déclaration

```
# Création d'un set vide
mon_set = set()

# Création d'un set avec des éléments
mon_set = {1, 2, 3, 4}

# Les éléments en double sont automatiquement éliminés
mon_set = {1, 2, 2, 3, 4}  # Résultats : {1, 2, 3, 4}
```

Les ensembles

Les méthodes

```
# Création de sets
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Ajouter un élément
set1.add(6)

# Supprimer un élément
set1.discard(2)

# Vérifier l'appartenance
print(4 in set2)  # True
```

Les ensembles

Les méthodes

```
# Opérations ensemblistes

union = set1.union(set2)
union = set1 | set2

intersection = set1.intersect(set2)
intersection = set1 & set2

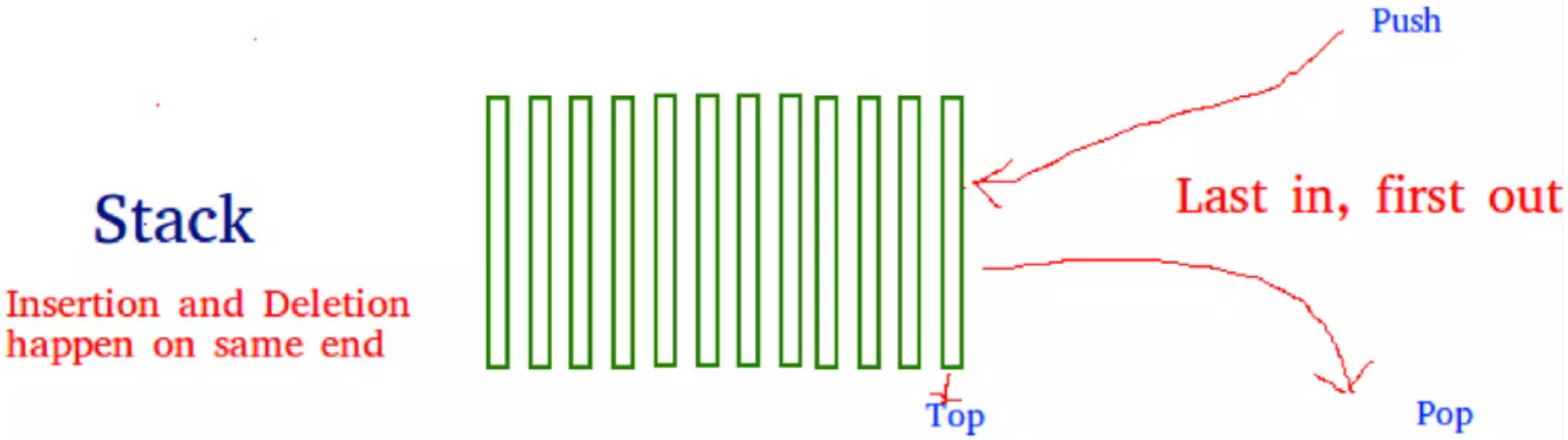
difference = set1.minus(set2)
difference = set1 - set2

difference_symetrique = set1. symmetric_difference(set2)
difference_symetrique = set1 ^ set2
```

Les Piles

Définition

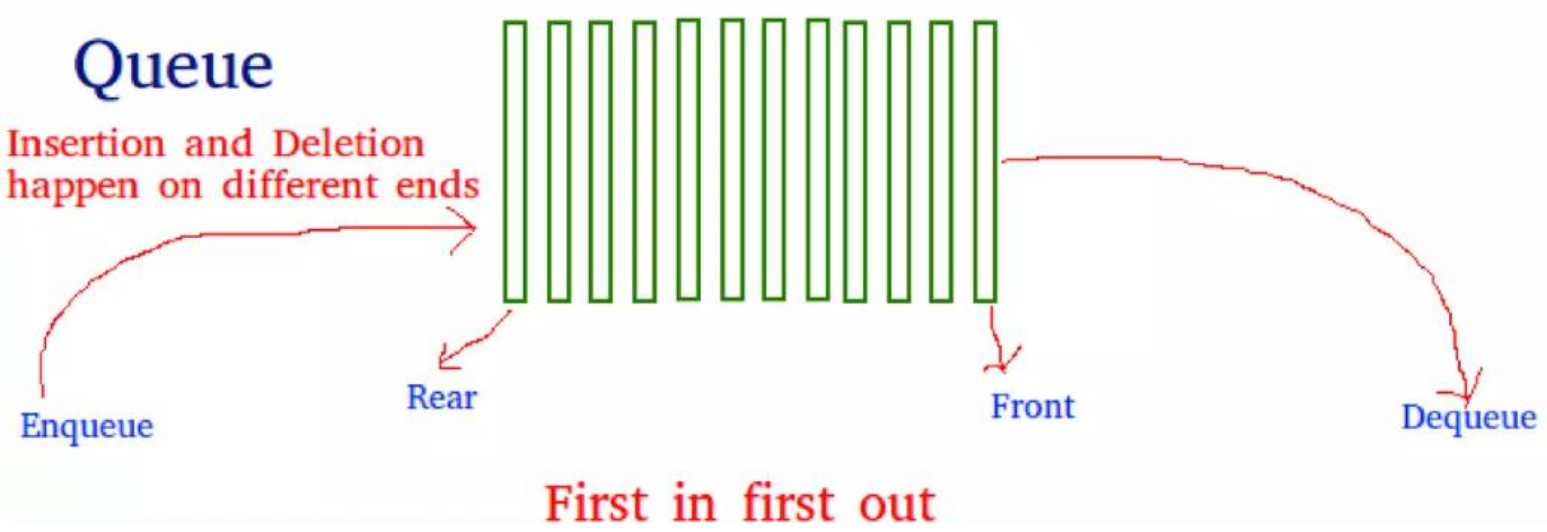
Une pile (ou stack en anglais) est une structure de données qui suit le principe LIFO (Last In, First Out), c'est-à-dire que le dernier élément ajouté est le premier à être retiré. Python ne fournit pas de structure de données dédiée pour les piles, mais on peut facilement implémenter une pile en utilisant des listes ou la classe **deque** du module **collections**.



Les Files

Définition

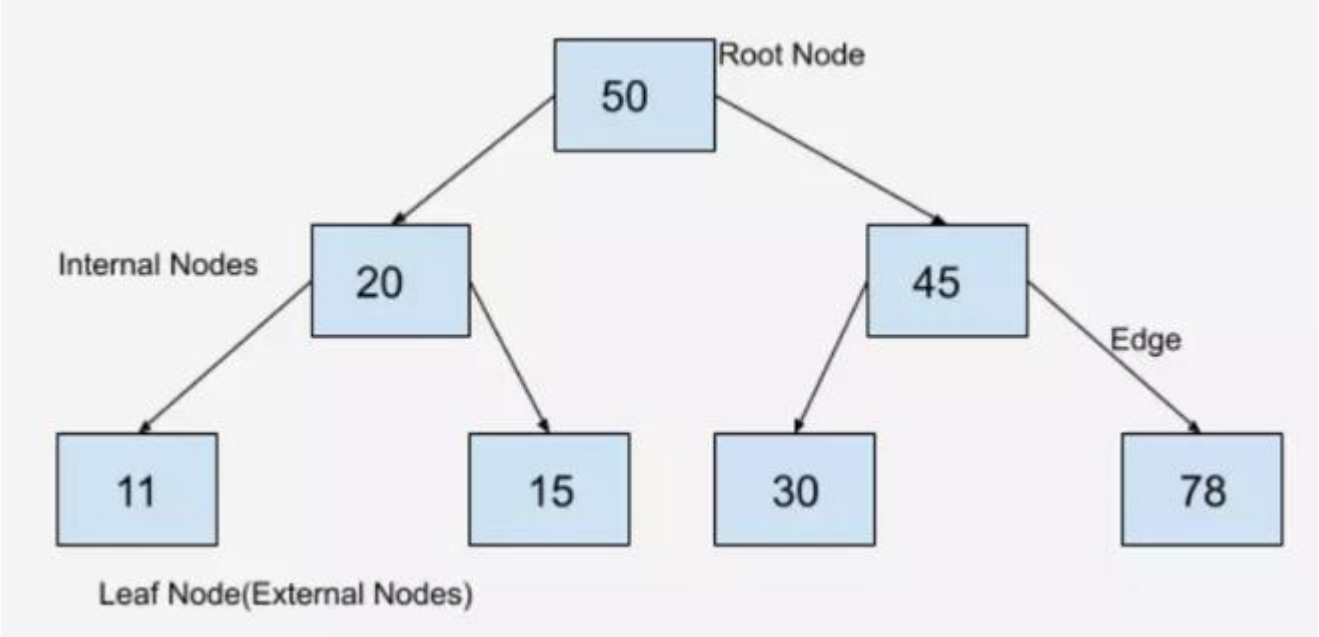
une file (ou queue en anglais) est une structure de données qui suit le principe FIFO (First In, First Out), ce qui signifie que le premier élément ajouté à la file est le premier à en être retiré. Contrairement aux piles (stacks), où le dernier élément ajouté est le premier à sortir, les files sont utilisées pour modéliser des systèmes où les éléments doivent être traités dans l'ordre de leur arrivée.



Les Arbres

Définition

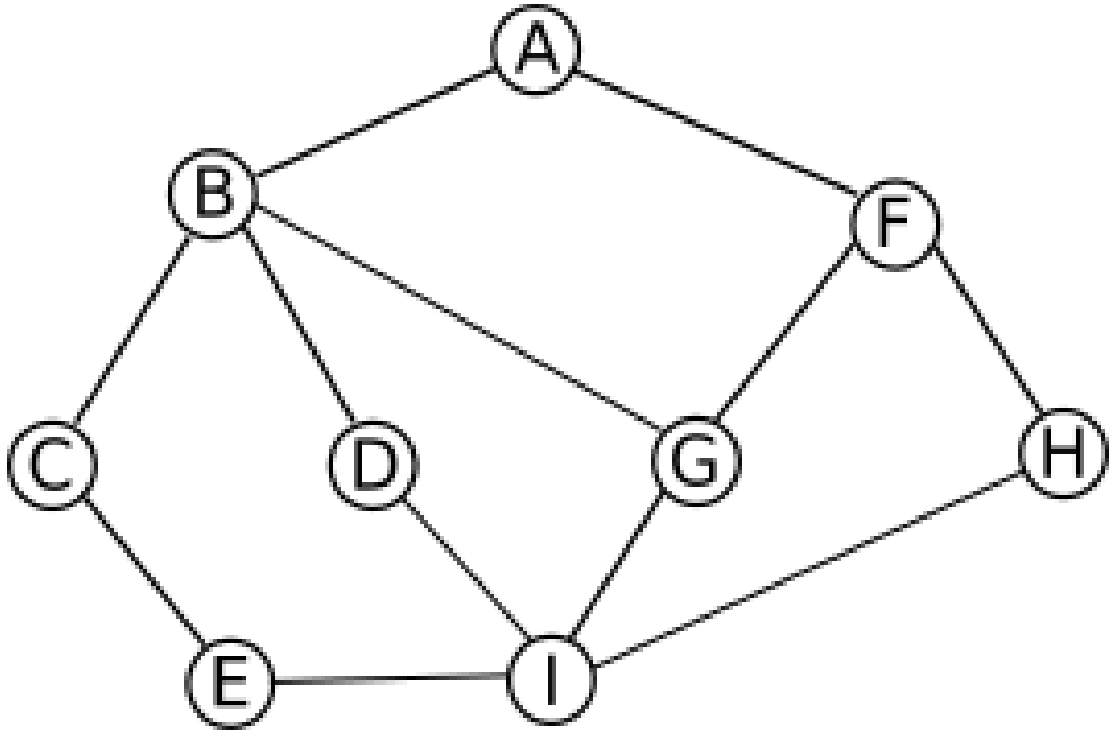
un **arbre** (ou **tree** en anglais) est une structure de données hiérarchique qui se compose de **nœuds** (ou **nodes**) reliés entre eux par des **arêtes** (ou **edges**). Les arbres sont utilisés pour représenter des relations hiérarchiques telles que des structures de dossiers, des organisations, des expressions arithmétiques, et bien plus encore.



Les Graphes

Définition

Un graphe est une structure de données composée de nœuds (ou sommets) et de liens (ou arêtes) qui relient ces nœuds. Les graphes sont utilisés pour représenter des relations entre des objets.



Comparaison

Le choix de la structure de données pour résoudre un problème dépend de plusieurs facteurs, notamment la nature des opérations que vous devez effectuer, les contraintes de performance, et les caractéristiques spécifiques de vos données. Voici quelques critères clés à considérer pour choisir la structure de données la plus adaptée :

Comparaison

Type d'Opérations :

Recherche : Si vous avez besoin d'effectuer de nombreuses recherches rapides, les hashmaps (dictionnaires en Python) ou les ensembles sont souvent appropriés.
Insertion/Suppression : Les listes chaînées sont efficaces pour les insertions et suppressions fréquentes, tandis que les tableaux dynamiques (comme les listes en Python) sont efficaces pour les accès aléatoires.
Accès aléatoire : Les tableaux dynamiques ou les listes sont souvent utilisés pour un accès rapide à des éléments à des positions spécifiques.
Ordre des Éléments : Si l'ordre des éléments est important et doit être conservé, les listes ou les structures comme les OrderedDict en Python (avant Python 3.7) sont préférables.

Comparaison

Complexité Temporelle :

Tableaux (Listes en Python) : Accès rapide $O(1)$ par index, mais les insertions et suppressions peuvent être coûteuses si elles ne se font pas à la fin $O(n)$.

Listes Chaînées : Bonnes performances pour les insertions et suppressions $O(1)$ à un endroit donné, mais les accès par index sont $O(n)$.

Hashmaps/Dictionnaires : Accès, insertion, et suppression en moyenne $O(1)$, mais peuvent souffrir de collisions et ne conservent pas l'ordre des éléments (dans les versions antérieures à Python 3.7).

Arbres Binaires de Recherche : Accès, insertion, et suppression en $O(\log n)$ en moyenne, et maintiennent un ordre partiel des éléments.

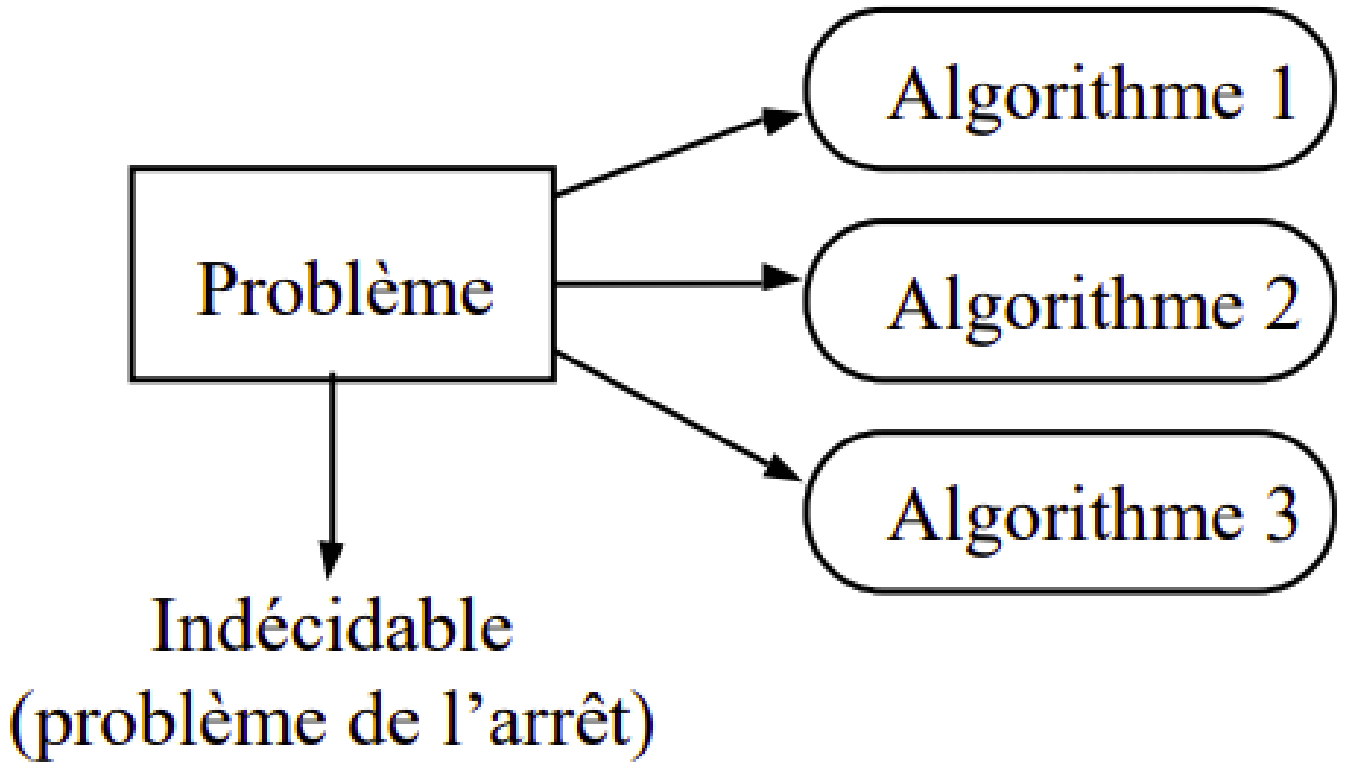
FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

DÉPARTEMENT DE MATHÉMATIQUES ET INFORMATIQUE

Complexité algorithmique

Master : MNSA

Pr: Youssef Ouassit


Problème
Algorithme 1
Algorithme 2
Algorithme 3
Indécidable
(problème de l'arrêt)

La suite Fibonacci :

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_{n+2} = U_{n+1} + U_n \end{cases}$$

```
1  def fibIt(n):
2      a, b = 0, 1
3      for i in range(0, n):
4          a, b = b, a + b
5      return a
```

La suite Fibonacci :

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_{n+2} = U_{n+1} + U_n \end{cases}$$

```
1  def fibRec(n):
2      if n==0 || n==1 :
3          return n
4      return fibRec(n-1) + fibRec(n-2)
```

La suite Fibonacci :

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_{n+2} = U_{n+1} + U_n \end{cases}$$

```
1     F = [0]*50 #array to store fibonacci terms
2
3   def fibonacci_bottom_up(n):
4       F[n] = 0
5       F[1] = 1
6
7       for i in range(2, n+1):
8           F[i] = F[i-1] + F[i-2]
9       return F[n]
```

La suite Fibonacci :

$$\begin{cases} U_0 = 0 \\ U_1 = 1 \\ U_{n+2} = U_{n+1} + U_n \end{cases}$$

```
1     F = [-1]*50 #array to store fibonacci terms
2
3   def fibonacci_top_down(n):
4     if (F[n] < 0):
5       if (n==0):
6         F[n] = 0
7       elif (n == 1):
8         F[n] = 1
9       else:
10        F[n] = fibonacci_top_down(n-1) + fibonacci_top_down(n-2)
11    return F[n]
```

Quel est le meilleur algorithme ?

Quel est le critère de comparaison ?

Comment comparer deux algorithmes ?

- La complexité d'un problème P est la mesure des quantités de ressources nécessaire à la résolution du problème P.
- Cette résolution est basée sur une estimation du **nombre d'opérations élémentaires effectuées** par l'algorithme en fonction **de la taille de données en entrée** de l'algorithme.

L'efficacité d'un algorithme peut être évalué en temps et en espace :

- **complexité spatiale** : permet de quantifier l'utilisation de la **mémoire**
- **complexité temporelle** : permet de quantifier la **vitesse** d'exécution

On s'intéresse essentiellement à la complexité en temps (ce qui n'était pas forcément le cas quand les mémoires coutaient cher)

L'évaluation de la complexité peut se faire à plusieurs niveaux :

- au niveau de l'exécution du programme expérimentalement
- au niveau purement algorithmique, par l'analyse et le calcul

Méthode expérimentale :

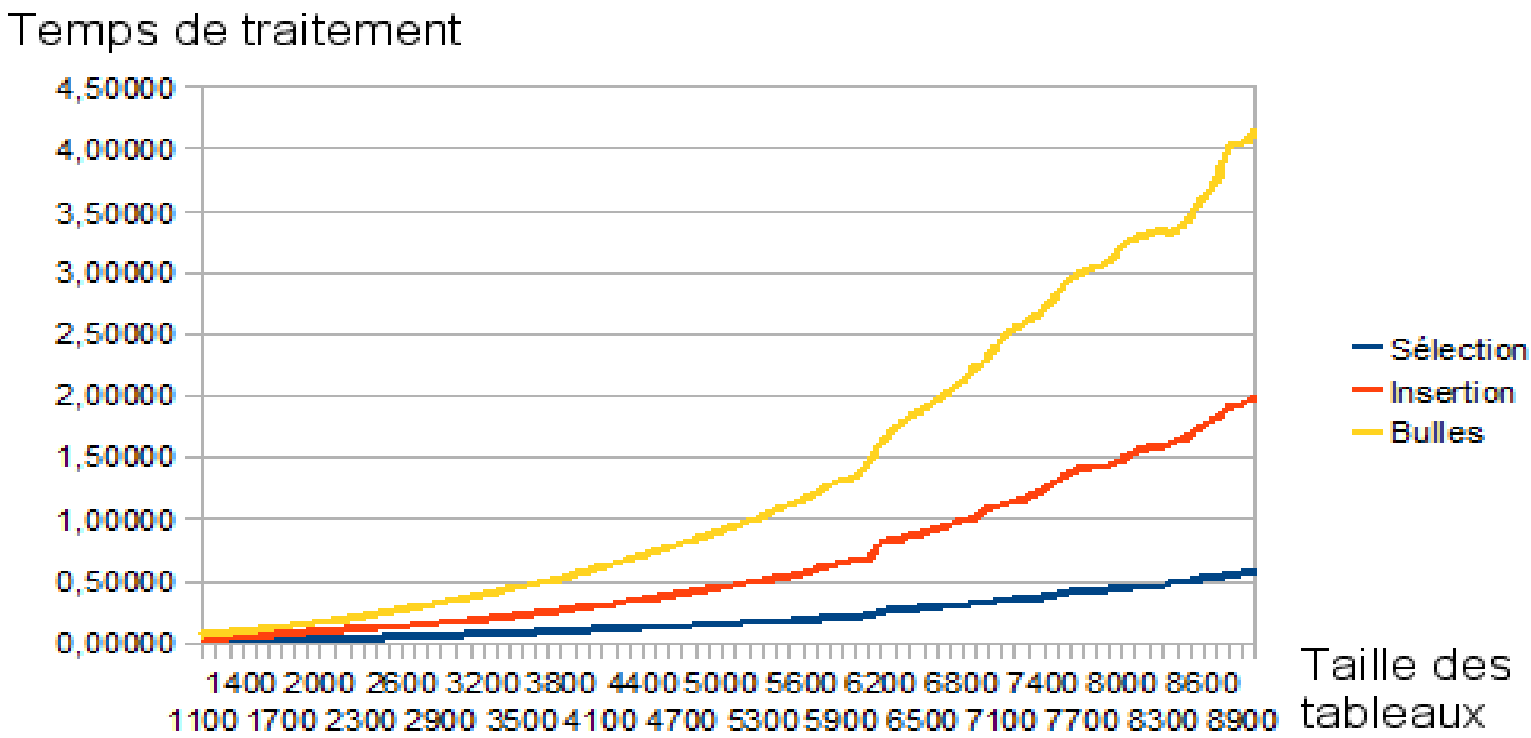
- La solution la plus intuitive pour choisir la meilleure solution consiste a mesuré le temps d'exécution exacte pour chaque algorithme.
- Pour ce faire, python offre le module time qui permet de calculer le temps d'exécution.

```
from time import time

t0 = time( )*1000
tri_fusion(L)
t1 = time( )*1000
print(t1-t0)
```

Méthode expérimentale

- Il est possible d'évaluer de façon expérimentale le temps d'exécution des programmes.



Méthode expérimentale

Inconvénients :

- Implémentation de l'algorithme
- Résultats dépends de l'environnements.
- Lancer plusieurs exécutions avec des tailles de données différents: beaucoup de temps

Méthode expérimentale

⇒**Environnement de simulation**

- Processeur Intel Core i5-4210U CPU 1,70GHz x 4
- Mémoire vive 8Go
- OS : Linux Ubuntu 14.04 LTS 64bits

⇒**Résultats de l'expérience**

N	10^2	10^3	10^4	10^5	10^6	10^7
Tri fusion	0,3ms	0,9ms	5,5ms	20ms	215ms	2,3s
Tri insertion	1,8ms	67ms	5,7ms	11mn	19h	66j



Méthode théorique

Avantages :

- Comparer les algorithmes théoriquement
- Pas besoin d'implémentation
- Taille des exemplaires n'est pas une contrainte

Etapes de calcul de la complexité

1. Identifier le(s) paramètre(s) de complexité de l'algorithme

2. Calculer le coût en fonction du paramètre de la complexité

3. Calculer la complexité asymptotique

Paramètre de complexité (1/5)

- Le paramètre de la complexité est ce qui va faire varier le temps d'exécution de l'algorithme
- Exemple : la calcul de la factorielle

```
def factoriel ( a ) :
        f=1
        for i in range(2, a ) :
                f = f*i
        return f
```

Le paramètre de complexité est la valeur de a

n = a

Paramètre de complexité (2/5)

- Exemple : multiplication des éléments d'une liste par un nombre

```
def multiplication(L,x):
    for i in range(len(L)):
        L[i]=L[i]*x
```

Le paramètre de complexité est la valeur de len(L)
n = len(L)

Paramètre de complexité (3/5)

Exemple : somme de la i ème ligne d'une matrice

Le paramètre de complexité est la valeur de len(M[0])
n = len(M[0])

```
def somme(M, i ):
    s=0
    for j in len(M[i]):
        s = s + M[i][j]
    return s
```

Paramètre de complexité (4/5)

Exemple : table de multiplication d'un entier

Pas de paramètre de complexité

```
def table1(n):
    for i in range(11):
        print(i * n)
```

Paramètre de complexité (5/5)

- Exemple : produit de deux matrices

Paramètres de complexité :
l = Nombre de lignes de M
c = Nombre de colonnes de M
p= Nombre de de colonnes de N

n = max(l, p, c)

```
def produit(M, N):
    l = len(M)
    c = len(M[0])
    p = len(N[0])
    R = [[0]*l for I in range(p) ]
    for i in range(l):
        for j in range(p):
            for k in range(c):
                R[i][j] += M[i][k]*N[k][j]
    Return R
```

Calculer le coût

Calculer le coût d'un algorithme

Pour déterminer le coût d'un algorithme, on se fonde en général sur le *modèle de complexité* suivant :

1. Une affectation, une comparaison, l'évaluation d'une opération arithmétique, les instructions de lecture et d'écriture ayant en général un faible temps d'exécution, celui-ci sera considéré comme l'unité de mesure du coût d'un algorithme.
2. Le coût des instructions p et q en séquence est la somme des coûts de l'instruction p et de l'instruction q.
3. Le coût d'un test **if b: p else: q** est inférieur ou égal au maximum des coûts des instructions *p* et q, plus le temps d'évaluation de l'expression b.
4. Le coût d'une boucle **for** : le coût est égal au nombre d'éléments de l'itérable multiplié par le coût de l'instruction p si ce dernier ne dépend pas de la valeur de i. Quand le coût du corps de la boucle dépend de la valeur de i, le coût total de la boucle est la somme des coûts du corps de la boucle pour chaque valeur de i.
5. Le cas des boucles **while** est plus complexe à traiter puisque le nombre de répétitions n'est en général pas connu a priori. On peut majorer le nombre de répétitions de la boucle de la même façon qu'on démontre sa terminaison et ainsi majorer le coût de l'exécution de la boucle.

Coût des instructions élémentaires

Exemple 1 :

```
A=5
B=A+5
print(A+B)
```

Nombre d'instructions élémentaires : 5

Coût des instructions élémentaires

Exemple 2 :

```
if i%2 == 0 :
      n = i//2
else:
      i = i+1
      n = i//2
```

Nombre d'instructions élémentaires : 6

Coût des instructions élémentaires

Exemple 3 :

```
for i in range(10):
    a = int(input())
    print(a*5)
```

Nombre d'instructions élémentaires : 50

Coût des instructions élémentaires

Exemple 3 :

```
i=1
 while i <= n :
        somme = somme + i
         i = i + 1
```

Nombre d'instructions élémentaires : 5n+1

Nuances de la complexité temporelles

Lorsque, pour une valeur donnée du paramètre de complexité, le temps d'exécution varie selon les données d'entrée, on peut distinguer :

- La complexité au pire : temps d'exécution maximum, dans le cas le plus défavorable.
- La complexité au mieux : temps d'exécution minimum, dans le cas le plus favorable (en pratique, cette complexité n'est pas très utile).
- La complexité moyenne : temps d'exécution dans un cas médian, ou moyenne des temps d'exécution.

Le plus souvent, on utilise la complexité au pire, car on veut borner le temps d'exécution.

Nuances de la complexité temporelles

- Exemple :

```
def rechercherElement(L , x) :
        for i in range(0, len(L)):
                if L[i]==x :
                        return True
        return False
```

Le coût au pire :
2n+1
Le coût au mieux:
2
Le coût au moyen:
n+1

Complexité asymptotique : La notation O

Remarque : L'analyse théorique de l'efficacité d'un algorithme se fait à une constante près pour ne pas tenir compte de :

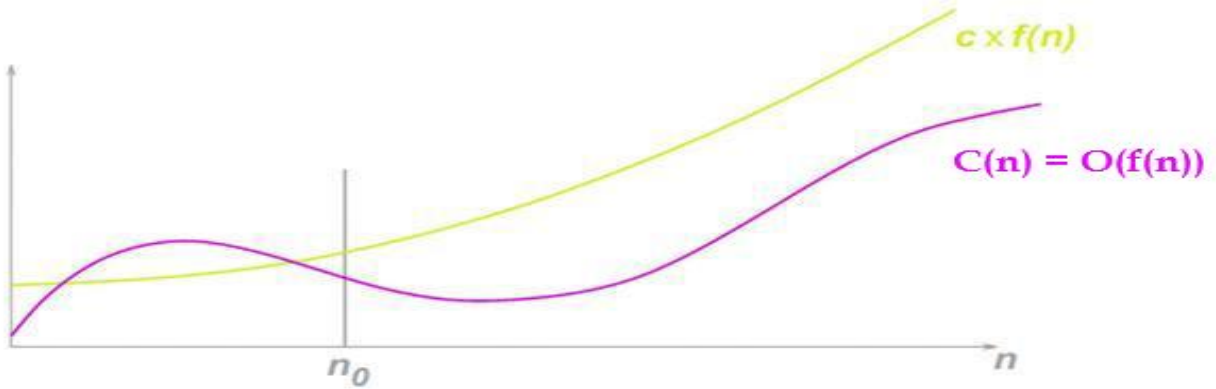
- langage de programmation
- compilateur et système d'exploitation
- puissance de l'ordinateur

Ce qui nous intéresse n'est pas un temps précis d'exécution mais un ordre de grandeur de ce temps d'exécution en fonction de la taille des données.

- **Première approximation :** Considérer la complexité au pire des cas
- **Deuxième approximation :** Considérer le calcule asymptotique de la complexité, s'intéresser juste au grandes quantités des données.

Complexité asymptotique : La notation O

- Soit C(n) une fonction qui désigne le temps de calcul d'un algorithme A.
- On dit que C(n) est en grand O de f(n) : C(n) = O(f (n)) si et seulement
- si : ∃(n0; c) telle que C(n) <= c*f (n) pour tous n >= n0



La notation O(f (n)) est aussi appelée Notation de Landau : (Complexité asymptotique) (i.e.quand n → ∞)

Notation O : Simplification

On calcule le temps d'exécution comme avant, mais on effectue les simplifications suivantes :

- On oublie les constantes multiplicatives (elles valent 1) ;
- On annule les constantes additives ;
- On ne retient que les termes dominants ;

- Soit un algorithme effectuant $C(n) = 4n^3 - 5n^2 + 2n + 3$ opérations ;
- => On remplace les constantes multiplicatives par 1 : $1n^3 - 1n^2 + 1n + 3$
- => On annule les constantes additives : $n^3 - n^2 + n + 0$
- => On garde le terme de plus haut degré : n^3
- Donc : $C(n) = O(n^3)$:

Notation O : Propriétés

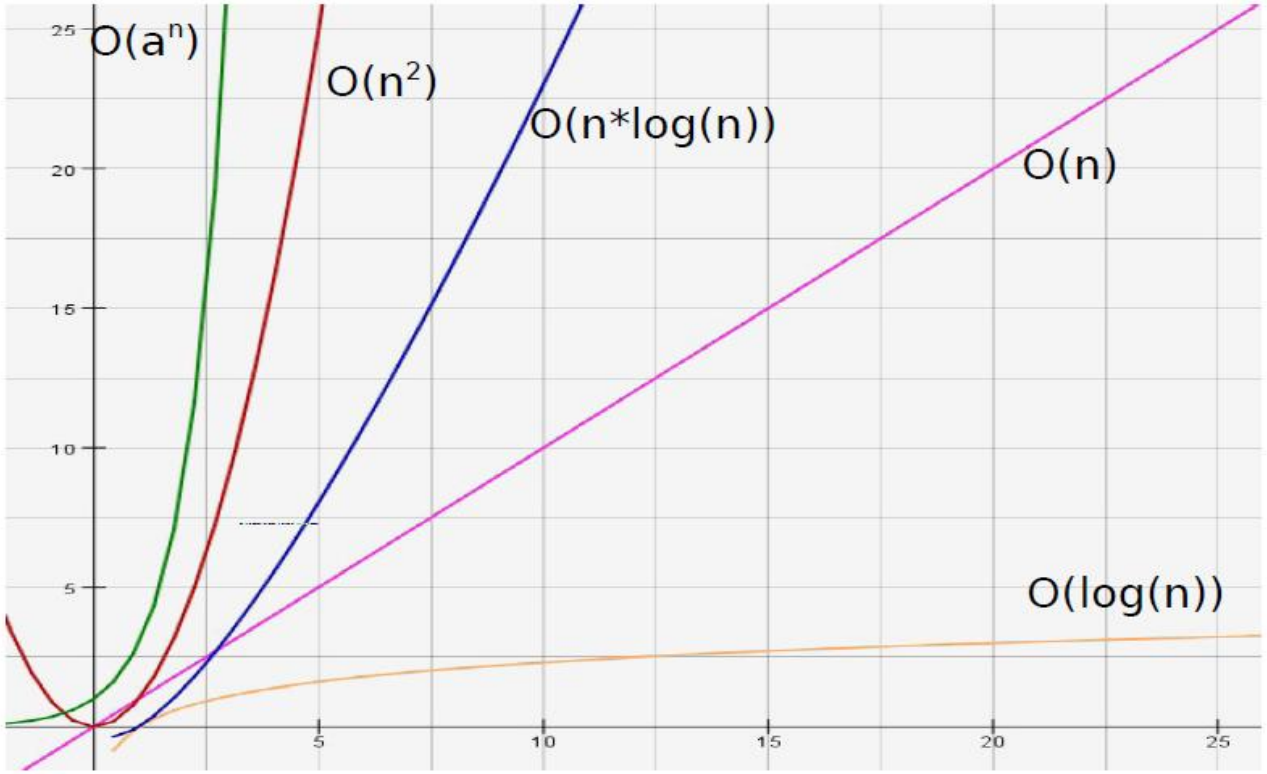
- c *O(f (n)) = O(f (n))
- O(f (n)) + O(g(n)) = O(f (n) + g(n))
- O(f (n)) * O(g(n)) = O(f (n) * g(n))

Classes de la complexité

- On voit souvent apparaitre les complexités suivantes :

Complexité	Nom courant	Description
$O(1)$	constante	Le temps d'exécution ne dépend pas des données traitées, ce qui est assez rare !
$O(log(n))$	logarithmique	augmentation très faible du temps d'exécution quand le paramètre croit.
$O(n)$	linéaire	augmentation linéraire du temps d'exécution quand le paramètre croit.
$O(nlog(n))$	quasi-linéaire	augmentation un peu supérieure à O(n)
$O(n^2)$	quadratique	quand le paramètre double, le temps d'exécution est multiplié par 4. Exemple : algorithmes avec deux boucles imbriquées.
$O(n^k)$	polynômiale	ici, n^k est le terme de plus haut degré d'un polynôme en n ; il n'est pas rare de voir des complexités en $O(n^3)$ ou $O(n^4)$.
$O(k^n)$	exponentielle	quand le paramètre double, le temps d'exécution est élevé à la puissance k avec $k > 1$.
$O(n!)$	factorielle	asymptotiquement équivalente à n^n

Classes de la complexité



Exemples

```
def factoriel(n):
    f=1
    for i in range(2, n+1):
        f=f*i
    return f
```

Complexité linéaire : O(n)

Exemples

```
def minimum(L):
    min = L[0]
    for i in range(0, len(L)):
        if L[i] < min :
            min = L[i]
    return min
```

Complexité linéaire : O(n)

Exemples

```
def table_multiplication(n):
    for i in range(1, 11):
        print(n,"x",i,"=",n*i)
```

Complexité constante : O(1)

Exemples

Somme des éléments de la diagonale d'une matrice carrée :

```
def   sommeMat1(A):
        s=0
        n = len(A)
        for i in range(n):
                for j in range(n):
                        if i == j : s= s + A[i][j]
        return s
def   sommeMat2(A):
        n = len(A)
        s=0
        for i in range(n):
                s = s + A[i][i]
        return s
```

Complexité quadratique : $O(n^2)$

Complexité linéaire : O(n)

Exemples

```
def tri_bulle(L) :
    for i in range(len(L)-1, 0, -1 ) :
        for j in range(0, i):
            if L[j]>L[j+1] :
                L[j],L[j+1] = L[j+1], L[j]
    return L
```

Complexité quadratique: $O(n^2)$

Paramètre de complexité : n=len(L)

$C(n) = \sum_{i=1}^{n-1} 6i + 1$

$= 6 \sum_{i=1}^{n-1} i + 1$

$= 6(n-1)(n)/2 + 1$

$= 3n^2 - 3n + 1$

Exemples

```
def tri_selection(T) :
    for i in range(len(T)-1) :
        posmin=i
        for j in range(i+1,len(T)) :
            if T[j]<T[posmin] :
                posmin=j
        T[i],T[posmin]=T[posmin],T[i]
    return T
```

Complexité quadratique: $O(n^2)$

Paramètre de complexité : n=len(T)

$C(n) = \sum_{i=0}^{n-2}(3 + (\sum_{j=i+1}^{n-1} 2))$

$= \sum_{i=0}^{n-2} 3 + \sum_{i=0}^{n-2}(\sum_{j=i+1}^{n-1} 2)$

$= 3(n-1) + \sum_{i=0}^{n-2}(\sum_{j=1}^{n-i-1} 2)$

$= 3(n-1) + 2\sum_{i=0}^{n-2}(n-i-1)$

$= 3(n-1) + 2(\sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1)$

$= 3(n-1) + 2(n(n-1) - (n-1)(n)/2 - (n-1)) = n^2 - 5n - 2$

Exemples

```
def tri_insertion(T) :
    for i in range(1,len(T)) :
        k=i
        while k>0 and T[k]<T[k-1] :
            T[k], T[k-1] = T[k-1], T[k]
            k = k – 1
```

Complexité quadratique: $O(n^2)$

Paramètre de complexité : n=len(T)

$C(n) = \sum_{i=1}^{n-1}(1 + 10i)$

$C(n) = (n-1) + 10((n-1)(n)/2)$

$C(n) = 5n^2 - 4n - 6$

Exemples

```
def RechDichotomique(T,x) :
    inf=0
    sup=len(T)-1
    while inf<sup :
        m=(inf+sup)//2
        if T[m]==x :
            return m
        if T[m]<x :
            inf=m+1
        else :
            sup=m-1
    return -1
```

Complexité logarithmique : $O(\log_2(n))$

Complexité des fonctions récursives

Comment calculer la complexité des fonctions récursives ?

```
def  factoriel(n) :
     if n==0 or n==1 :
          return 1
     return n*factoriel(n-1)
```

Le paramètre de complexité : n

C(0)=4

C(1)=4

C(n)=4 +3+ C(n-1) = C(n-1) + 7

Le terme général de la suite nous donne :

7n+4 = O(n)

Complexité des fonctions récursives

Equation : $c(n) = c(n-1) + b$
Solution : $c(n) = c(0) + b*n = O(n)$
Exemples : factorielle, recherche séquentielle récursive dans un tableau

Equation : $c(n) = a*c(n-1) + b$, $a \neq 1$
Solution : $c(n) = a^n*(c(0) - b/(1-a)) + b/(1-a) = O(a^n)$
Exemples : répétition a fois d'un traitement sur le résultat de l'appel récursif

Equation : $c(n) = c(n-1) + a*n + b$
Solution : $c(n) = c(0) + a*n*(n+1)/2 + n*b = O(n^2)$.
Exemples : traitement en coût linéaire avant l'appel récursif, tri à bulle

Equation : $c(n) = c(n/2) + b$
Solution : $c(n) = c(1) + b*\log_2(n) = O(\log(n))$
Exemples : élimination de la moitié des éléments en temps constant avant l'appel récursif, recherche dichotomique récursive

Complexité des fonctions récursives

Equation : $c(n) = a*c(n/2) + b$, $a \neq 1$

Solution : $c(n) = n^{log2(a)} * (c(1) - b/(1-a)) + b/(1-a) = O(n^{log2(a)})$

Exemples : répétition a fois d'un traitement sur le résultat de l'appel récursif dichotomique

Equation : $c(n) = c(n/2) + a*n + b$

Solution : $c(n) = O(n)$

Exemples : traitement linéaire avant l'appel récursif dichotomique

Equation : $c(n) = 2*c(n/2) + a*n + b$

Solution : $c(n) = O(n*log(n))$

Exemples : traitement linéaire avant double appel récursif dichotomique, tri fusion

Complexité des fonctions récursives

```
def longueur(ch):
    if ch=="":
        return 0
    return 1+longueur(ch[1:])
```

C(n) = 2 + 2 + C(n-1)
C(n) = C(n-1) + 4
Complexité logarithmique : O(n)

Complexité des fonctions récursives

```
def RechDichotomiqueRec(T,x,i,j) :
    m=(i+j)//2
    if T[m]==x :
        return m
    if i>j :
        return -1
    if T[m]<x :
        return RechDichotomiqueRec(T,x,m+1,j)
    else :
        return RechDichotomiqueRec(T,x,i,m-1)
```

C(n) = 3 + 2 + 2 + 1 + 1 + C(n/2)
C(n) = C(n/2) + 9
Complexité logarithmique : $O(\log_2(n))$

Complexité des fonctions récursives

```
def fusionner(A, B):
    R = []
    while A!=[] and B!=[]:
        if A[0] < B[0] :
                R.append(A.pop(0))
        else :
                R.append(B.pop(0))
    return R + A + B

def tri_fusion(L):
    n = len(L)
    if n==0 or n==1:
        return L
    m=(n-1)//2
    return fusionner(tri_fusion(L[:m+1]),  tri_fusion(L[m+1:]))
```

C(n)=(n-1)*(3 + 3) + 4=6n-2

C(n) = C(n/2) + C(n/2) + 3n + 8

Complexité quasi logarithmique : O(n $\log_2(n)$)

Complexité des fonctions récursives

```
def partition(L, debut, fin):
    pivot = L[debut]
    ipivot = debut
    for i in range(debut+1, fin+1):
        if L[i]<=pivot :
                a = L.pop(i)
                L.insert(debut,a)
                ipivot = ipivot + 1
    return ipivot

def quick(L, debut, fin):
    if debut<fin :
        ipivot = partition(L, debut, fin)
        quick(L,debut,ipivot-1)
        quick(L,ipivot+1, fin)
```

Complexité de la fonction partition : O(n)

C(n) = 2 + O(n) + C(n-1) + O(1)
C(n) = C(n-1) + O(n)
Complexité quadratique au pire : $O(n^2)$

Calculer directement en notation asymptotique

```
def factoriel(n):
    f=1                              O(1)
    for i in range(2, n+1):          O(n) x  O(1)
        f = f * i
    return f                         O(1)
```

Complexité linéaire : O(n)

Calculer directement en notation asymptotique

```
def tri_bulle(L) :
    for i in range(len(L)-1, 0, -1 ) :
        for j in range(0, i):
            if L[j]>L[j+1] :
                L[j],L[j+1] = L[j+1], L[j]
    return L
```

O(n)

O(n) x O(1)

O(1)

Complexité linéaire : $O(n^2)$

Complexité : Manipulation de base d'une liste

liste est une liste de taille N

Instruction	Exemple	Complexité
Récupérer la longueur	len(liste)	O(1)
Ajouter un élément à la fin	liste.append(5)	O(1)
Accéder à un élément	liste[20]	O(1)
Remplacer un élément	liste[30]=100	O(1)
Parcourir une liste	for a in liste : print(a)	O(N)

Complexité : Manipulation de base d'une liste

liste est une liste de taille N

Instruction	Exemple	Complexité
Supprimer le dernier élément	liste.pop(-1)	O(1)
Supprimer le premier élément	liste.pop(0)	O(N)
Comparer deux listes	if liste == liste2 : print("listes égales")	O(N)
Faire une copie	copie = liste.copy()	O(N)
Trouver le min	min(liste)	O(N)
Trouver le max	max(liste)	O(N)
Vérifier si un élément est dans la liste	if 40 in liste : print("OK")	O(N)

Complexité : Manipulation de base d'un ensemble

E est un ensemble de taille N

Instruction	Exemple	Complexité
Récupérer la longueur	len(E)	O(1)
Ajouter un élément à la fin	E.add(5)	O(1)
Supprimer une élément	E.remove(20)	O(1)
Parcourir un ensemble	for a in E : print(a)	O(N)
Comparer deux ensembles	if E==autre : print("Ensembles égaux")	O(N)

Complexité : Manipulation de base d'un ensemble

E est un ensemble de taille N

Instruction	Exemple	Complexité
Faire une copie	copie = E.copy()	O(N)
Trouver le min	min(E)	O(N)
Trouver le max	max(E)	O(N)
Vérifier si un élément est dans l'ensemble	if 40 in E : print("OK")	O(1)

Complexité : Manipulation de base d'un dictionnaire

D est un dictionnaire de taille N

Instruction	Exemple	Complexité
Récupérer la longueur	len(D)	O(1)
Ajouter un élément à la fin	D[8] = 20	O(1)
Accéder à un élément si on connaît la clé	D[8]	O(1)
Parcourir une liste	for key, value in liste.items() : print(key,'=',value)	O(N)

Complexité : Manipulation de base d'un dictionnaire

D est un dictionnaire de taille N

Instruction	Exemple	Complexité
Supprimer un élément	D.pop(key)	O(1)
Comparer deux dictionnaires	if D == D2 : print("dictio égales")	O(N)
Faire une copie	copie = D.copy()	O(N)
Trouver le min	min(D)	O(N)
Trouver le max	max(D)	O(N)
Vérifier si une clé est dans la liste	if 40 in D.keys() : print("OK")	O(1)
Vérifier si une vlauer est dans la liste	if 40 in D.values() : print("OK")	O(N)

Complexité des fonctions récursives

Diviser pour régner : Théorème maître

Les algorithmes de type « diviser pour régner » résolvent a sous problèmes de taille n/b et combinent ensuite ces réponses en un temps $O(n^d)$, pour a,b,d > 0

Leur temps d'exécution C(n) peut donc s'écrire : $C(n) = aC(n/b) + O(n^d)$

(« n/b » signifiant ici partie entière inférieure ou supérieure de n/b)

Le terme général est alors :

$$\mathbf{C}(n) = \begin{cases} O(n^d) & \text{si } d > \log_b a \\ O(n^d \log n) & \text{si } d = \log_b a \\ O(n^{\log_b a}) & \text{si } d < \log_b a \end{cases}$$

Complexité des fonctions récursives

Diviser pour régner : Théorème maître

Leur temps d'exécution C(n) peut donc s'écrire : C(n) = aC(n/b) + O(n^d)

$$\mathbf{C}(n) = \begin{cases} O(n^d) & \text{si } d > \log_b a \\ O(n^d \log n) & \text{si } d = \log_b a \\ O(n^{\log_b a}) & \text{si } d < \log_b a \end{cases}$$

Algorithme	Relation de récurrence	Théorème Master	Complexité
Recherche par dichotomie	C(n) = C(n/2) + O(1)	a=1; b=2; d=0	O(log(n))
Tri fusion	C(n) = 2 C(n/2) + O(n)	a=2; b=2; d=1	O(n log(n))
Recherche de la médiane	C(n) = C(n/2) + O(n)	a=1; b=2; d=1	O(n)

FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

DÉPARTEMENT DE MATHÉMATIQUES ET INFORMATIQUE

Programmation orientée Objet : Les Objets, Les classes, les attributs et les méthodes

Master : MNSA

Pr: Youssef Ouassit

Introduction

1. **Introduction à la POO**
    1. Qu'est-ce que la POO ?
    2. Concepts Fondamentaux
2. **La POO en Python**

Introduction à la POO

Avertissement

Python intègre des particularités – pour ne pas dire bizarreries – que l'on ne retrouve pas dans les langages objets populaires (C++, Java, C#). Ex. création à la volée d'une propriété sur un objet (instance de classe), possibilité de l'utiliser dans les méthodes (alors qu'elle n'apparaît nulle part dans la définition de la classe)

Pour forger les bonnes habitudes, ce support adopte la présentation la plus conventionnelle possible par rapport aux autres langages précités, quitte à passer sous silence certaines possibilités de Python.

Introduction à la POO

Définition:

Un **paradigme de programmation** est un modèle ou une approche permettant de résoudre des problèmes en suivant un ensemble de principes ou de concepts. Chaque paradigme définit une façon particulière d'organiser et de structurer le code, influençant la manière dont les développeurs pensent et écrivent leurs programmes.
Différents paradigmes sont adaptés à différents types de problèmes et offrent diverses façons de gérer la complexité du logiciel.

Introduction à la POO

Principaux paradigmes de programmation :

- **Programmation impérative:**
  Ce paradigme se base sur une série d'instructions à exécuter dans un certain ordre. Le programme manipule l'état du système à l'aide de variables et d'assignations.
  Exemples de langages : C, Fortran, Python.
- **Programmation déclarative:**
  Dans ce paradigme, on décrit ce que le programme doit accomplir sans préciser comment l'accomplir. Cela inclut la programmation fonctionnelle et logique.
  Exemples de langages : HTML, SQL, Prolog.

Introduction à la POO

Principaux paradigmes de programmation :

- **Programmation fonctionnelle:**
  Ce paradigme traite le calcul comme une évaluation de fonctions mathématiques, sans état mutable ni effets de bord. Les fonctions sont des citoyens de première classe, et les transformations des données sont au cœur de cette approche
  Exemples de langages: Haskell, Lisp, Scala.
- **Programmation logique:**
  Basée sur des règles logiques, ce paradigme permet de définir un ensemble de faits et de règles, et le programme déduit des résultats par la logique. Ce type de programmation est couramment utilisé dans les systèmes experts.
  Exemples de langages: Prolog.

Introduction à la POO

Principaux paradigmes de programmation :

- **<u>Programmation orientée objet (POO)</u>**
  Organise le programme autour d'objets qui encapsulent les données (attributs) et les comportements (méthodes). Il est centré sur les concepts d'encapsulation, d'héritage, de polymorphisme, et d'abstraction.
  Exemples de langages : Java, C++, Python.

Introduction à la POO

Histoire de la POO:

- **Années 1960:** prémices de la POO avec le langage Simula-67
- **Années 1970-1980:** Smalltalk 72, puis Smalltalk 80 marquent le début effectif de la POO ainsi qué la pose de ses concepts de base.
- **A partir de 1980:** Prolifération des langages orientés objet ou hybrides :
  - 1980: Objective C
  - 1983: C++, Common LISP object
  - 1984: Eiffel
  - 1990: Python
  - 1996: Java 1
- **Années 1990:** évolutions du concept qui s'étend aux aspects conception, analyse (UML) et bases de données.

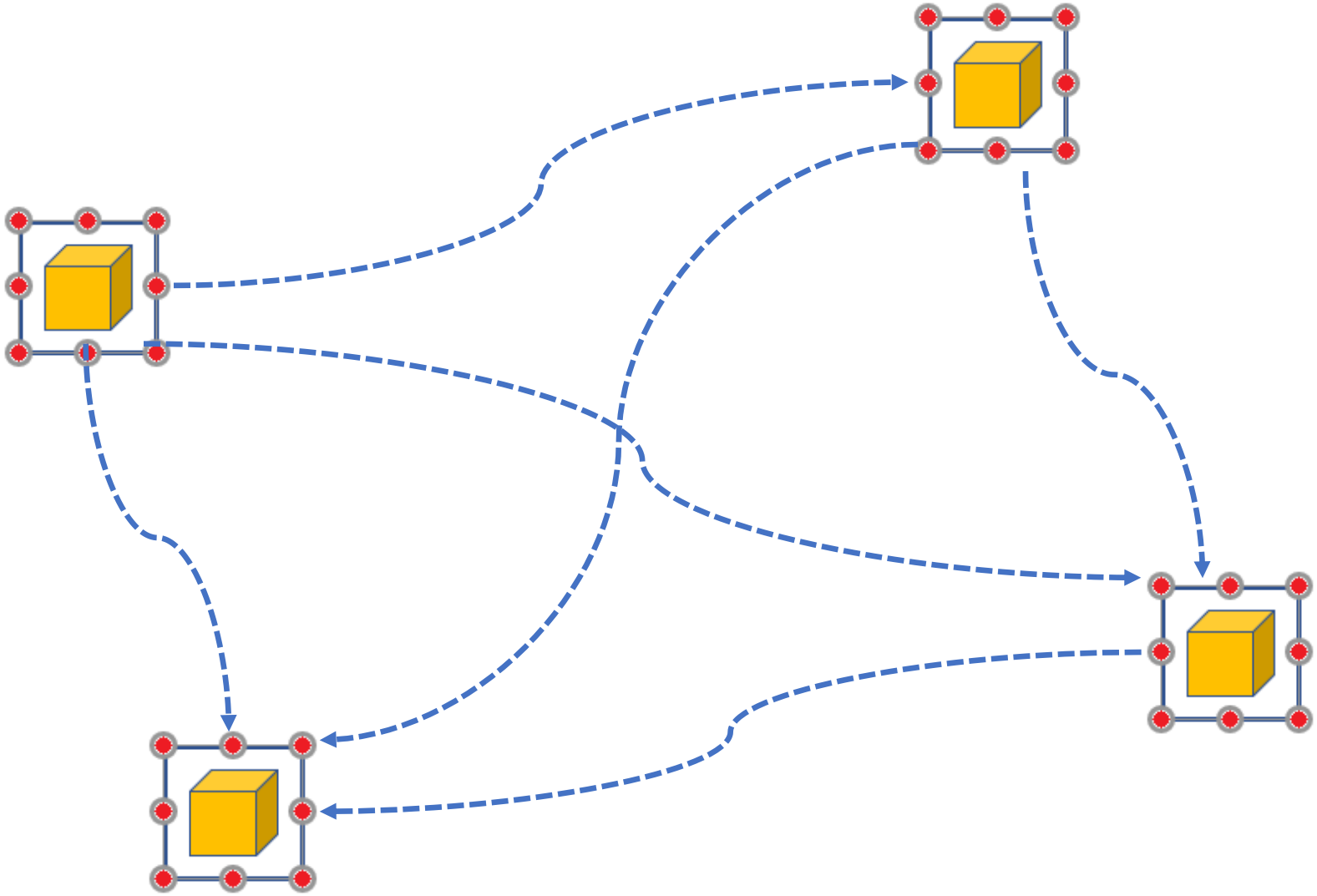
Introduction à la POO

Concepts de la POO:

- Class
- Objet
- Encapsulation
- Héritage
- Polymorphisme
- Abstraction

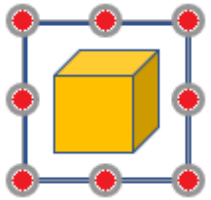
Introduction à la POO

Concepts de la POO:



Concepts de base

Qu'est ce qu'un Objet ?



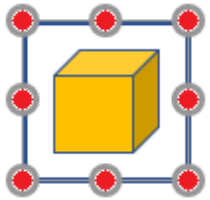
L'objet est une entité qui contient à la fois des données (sous forme **d'attributs**) et des comportements (sous forme de **méthodes**).

Caractéristiques d'un objet :

1. **Attributs** : Ce sont des variables qui stockent des informations propres à l'objet.
2. **Méthodes** : Ce sont des fonctions associées à un objet qui définissent son comportement.
3. **État** : L'ensemble des valeurs des attributs d'un objet représente son **état** à un moment donné.

Concepts de base

Qu'est ce qu'un Objet ?





C1

race = Siamois name = Kati hungry = 8 energy = 2 mood = 3
moew() sleep() play() feed()

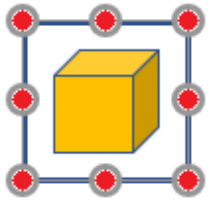
race = Persan name = Mika hungry = 2 energy = 9 mood = 8
moew() sleep() play() feed()



C2

Concepts de base

Qu'est ce qu'un Objet ?





C1

race = Siamois
name = Kati
hungry = 8
energy = 2
mood = 3

moew()
sleep()
play()
feed()

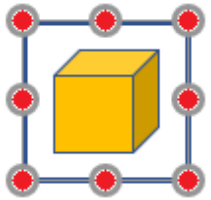
hungry +=1
energy +=1

energy - =-
mood +=1
meow()

mood += 1
energy += 1
hungry += 1

Concepts de base

Qu'est ce qu'un Objet ?





V1

marque = Lucid
annee = 2023
carburant = Electrique
vitesse = 0
niveau_carburant = 13%
Transmission = P

arreter()
demarrer()
accelerer()
ralentir()
charger()

vitesse = 0
transmission = P

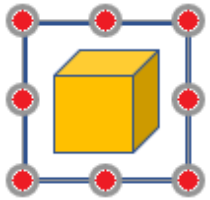
vitesse +=1
transmission +=1

vitesse -=1
transmission - =1

niveau_carburant +=1

Concepts de base

Qu'est ce qu'un Objet ?





V1

marque = Lucid
annee = 2023
carburant = Electrique
vitesse = 0
niveau_carburant = 13%
transmission = P

arreter()
demarrer()
accelerer()
ralentir()
charger()

nom = Mohamed
age = 50
ville = Casablanca
voiture = v1

manger()
dormir()
sedeplacer(v1)
changer_voiture()



P1

Concepts de base

Définition d'une classe

Une classe est **l'abstraction** d'un ensemble d'objet ayant une structure de **données commune** et disposant des **même méthodes**.

Une classe est un type permettant de regrouper dans la même structure : les informations (champs, propriétés, **attributs**) relatives à une entité ; les procédures et fonctions permettant de les manipuler (**méthodes**). Champs et méthodes constituent les membres de la classe.

Concepts de base

Définition d'une classe

Classe vs Objet ?

Termes techniques :

- « Classe » est la structure ;
- « Objet » est une instance de la classe (variable obtenue après instanciation) ;
- « Instanciation » correspond à la création d’un objet

Concepts de base

Définition d'une classe

Classe vs Objet ?

race name hungry energy mood
moew() sleep() play() feed()

Classe : Chat

instancier

instancier



C1

race = Siamois name = Kati hungry = 8 energy = 2 mood = 3
moew() sleep() play() feed()



C2

race = Persan name = Mika hungry = 2 energy = 9 mood = 8
moew() sleep() play() feed()

Concepts de base

Définition d'une classe

Exercice 1:

Créer une classe qui représente un nombre complexe
Ajouter à cette classe qui méthodes qui calcule :

- Le module
- Le conjugué
- La somme avec un autre complexe

Concepts de la POO

Définition d'une classe

Une classe est **l'abstraction** d'un ensemble d'objet ayant une structure de **données commune** et disposant des **même méthodes**.

Une classe est un type permettant de regrouper dans la même structure :

- les informations (**attributs**) relatives à une entité ;
- les procédures et fonctions permettant de les manipuler (**méthodes**).

Concepts de la POO

Définition d'une classe

Classe vs Objet ?

Termes techniques :

- « Classe » est la structure (C'est le type) ;
- « Objet » est une instance de la classe (variable obtenue après instanciation) ;
- « Instanciation » correspond à la création d'un objet

Concepts de la POO

Définition d'une classe

Classe vs Objet ?

race name hungry energy mood
moew() sleep() play() feed()

Classe : Chat

instancier

instancier



C1

race = Siamois name = Kati hungry = 8 energy = 2 mood = 3
moew() sleep() play() feed()



C2

race = Persan name = Mika hungry = 2 energy = 9 mood = 8
moew() sleep() play() feed()

Concepts de la POO

Définition d'une classe

Exercice 1:

Donner la définition d'une classe **Point**, un point est caractérisé par ses coordonnées.

Donner la définition d'une classe **Cercle** qui modélise un cercle. Un cercle est caractérisé par son centre et son rayon.

Ajouter des méthodes utilitaires aux deux classes.

Concepts de la POO

Définition d'une classe

Exercice 1: Solution

Point
- x : Réel - y : Réel
+ deplacer(dx: Réel, dy: Réel)

Cercle
- centre : Point - rayon : double
+ perimetre() + surface() + deplacer(dx: Réel, dy: Réel)

Concepts de la POO

L'encapsulation

Principe :

Les méthodes agissent exclusivement sur les données de l'objet.

L'encapsulation est un concept qui concerne les données (attributs) d'un objet.

L'encapsulation (définition): est ce concept de la POO qui interdit l'accès (en lecture/écriture) direct aux données d'un objet, mais qui le permet via des méthodes réservées à cet effet.

Concepts de la POO

L'encapsulation

Notion de visibilité :

- La visibilité sert à garantir la protection des données en :
  - en permettant de masquer les données et certaines méthodes les gérant,
  - et en laissant visibles d'autres méthodes devant servir à la gestion publique de l'objet.

Concepts de la POO

L'encapsulation

Type de visibilités :

- Publique (+)
- Privée (-)
- Protégée (#)
- Visibilité Package (~)

Concepts de la POO

L'encapsulation

Visibilité publique:

- Les attributs et méthodes sont publiques : Cela signifie qu'ils sont accessibles depuis tous les descendants et dans tous les modules.

- Les attributs et méthodes publiques n'ont pas besoin de restriction particulière.

Concepts de la POO

L'encapsulation

Visibilité privée:

- La visibilité privé restreint la porté d'un attribut ou d'une méthode au module où il/elle est déclaré(e).

- Ainsi, si un objet est déclaré dans une unité avec une donnée privée, alors cette donnée ne pourra être accédé qu'à l'intérieur même de l'unité.

Concepts de la POO

L'encapsulation

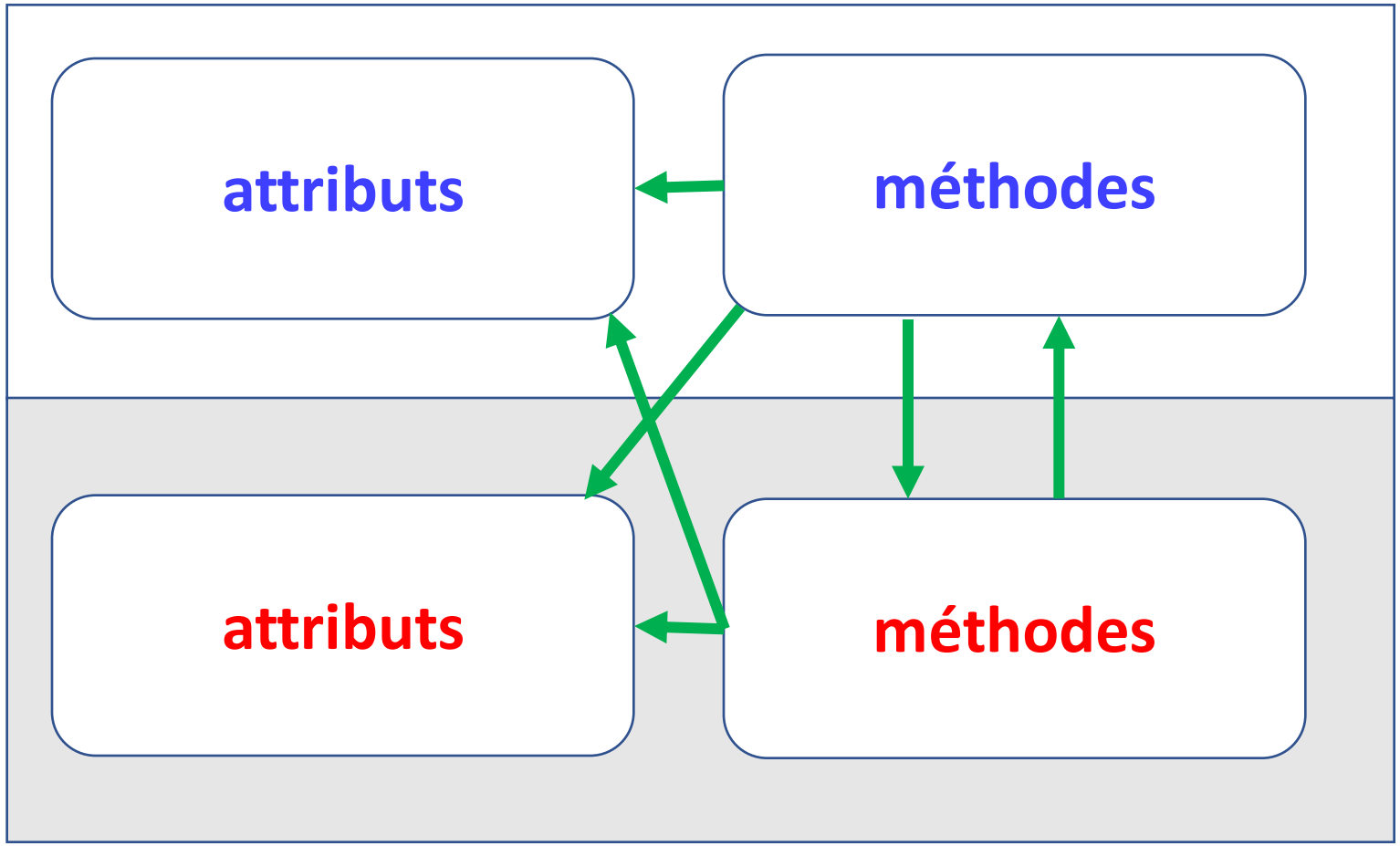
Visibilité protégée:

- La visibilité 'protégé' correspond à la visibilité 'privé' excepté que tout attribut ou méthode protégé(e) est accessible dans tous les descendants, quel que soit le module où il se situe.

Concepts de la POO

L'encapsulation

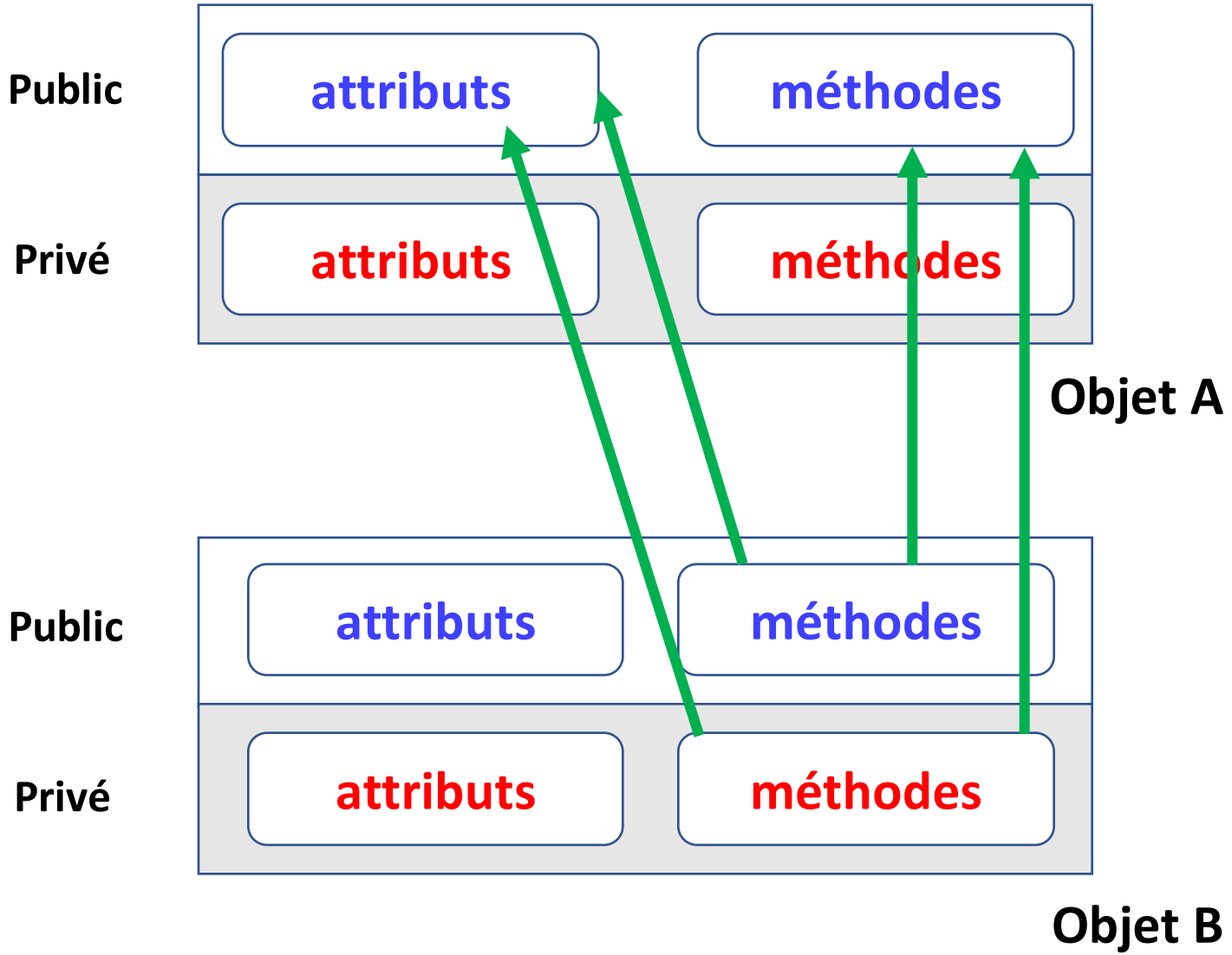
Au sein du même objet :



Concepts de la POO

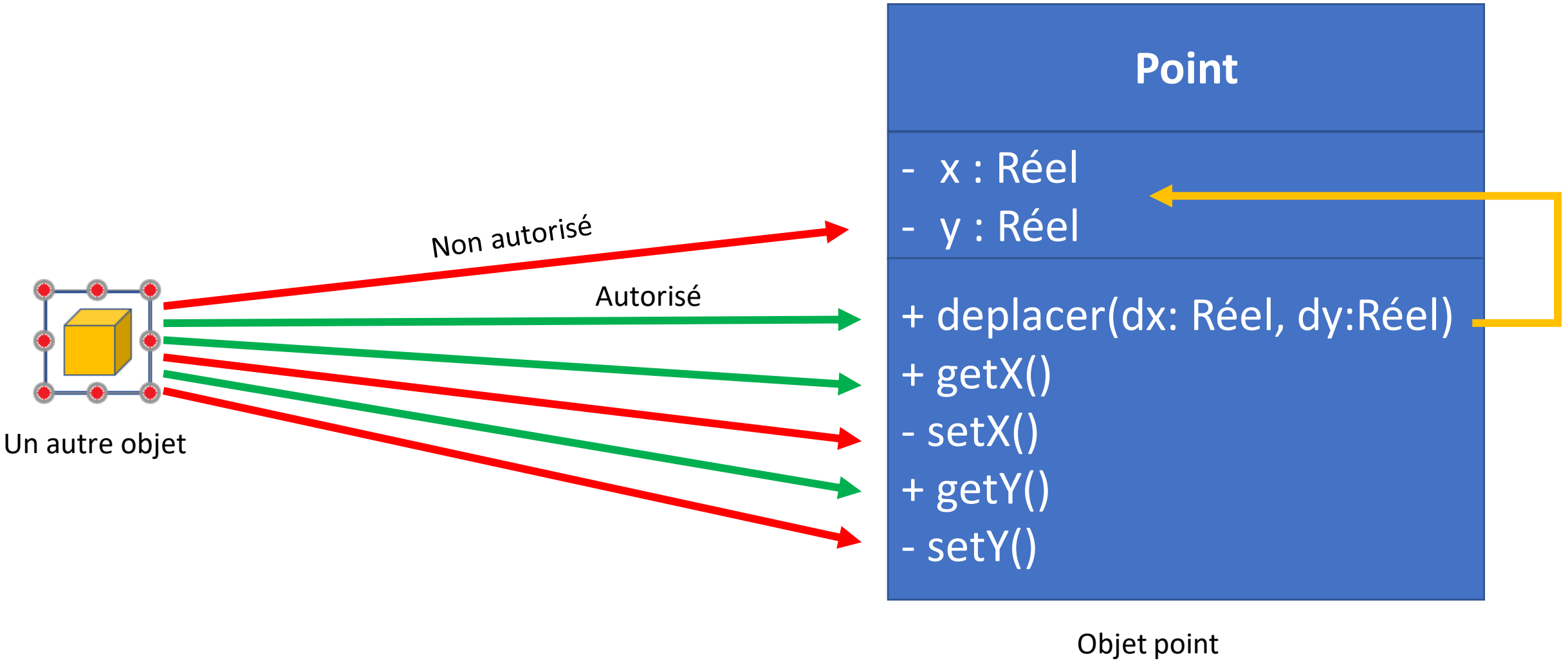
L'encapsulation

Entre deux objets différents :



Concepts de la POO

L'encapsulation



L'Héritage

Définition:

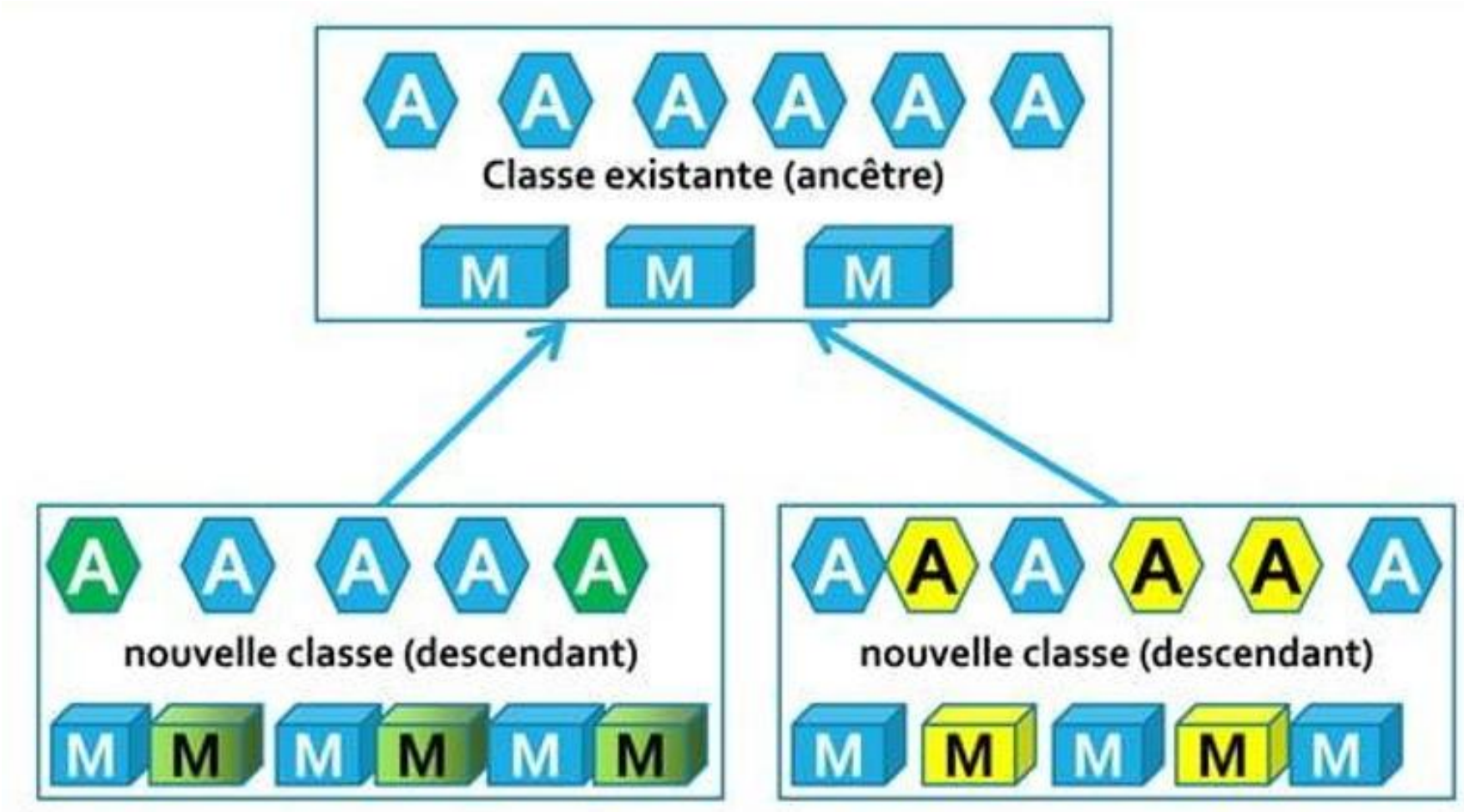
L'héritage est une manière de définir des abstractions de façon incrémentale:

- une définition B peut "hériter" d'une autre définition A.
- la définition B prend A comme base, avec éventuellement des modifications et des extensions.

L'encapsulation (définition): L'héritage en programmation orientée objet (POO) est un mécanisme qui permet à une classe (dite classe fille ou dérivée) d'hériter des propriétés et des méthodes d'une autre classe (dite classe mère ou base). Cela favorise la réutilisation du code et la création d'une hiérarchie entre les classes.

L'Héritage

Définition:



L'Héritage

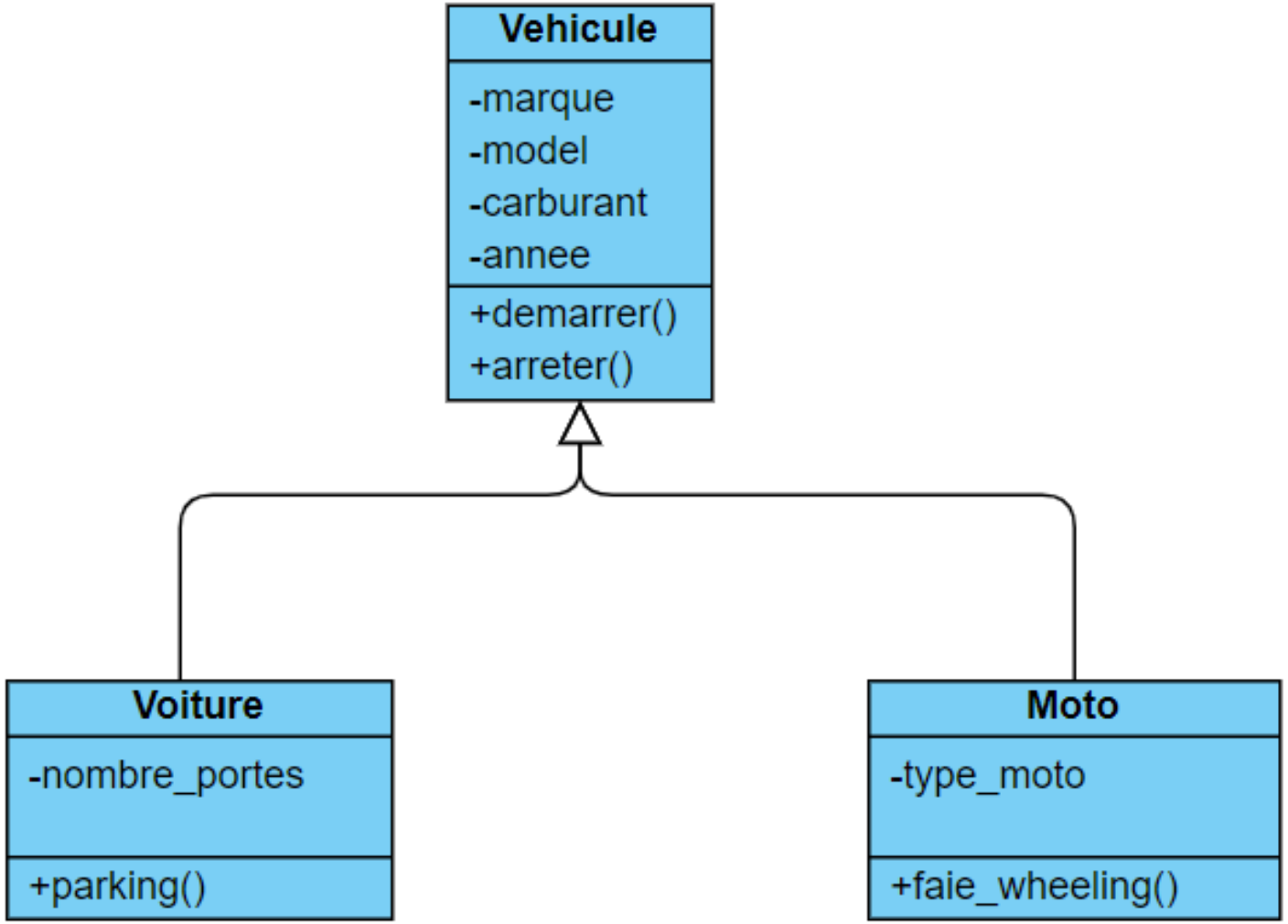
Exemples :

Pour chacun des abstractions suivantes trouver des exemples pour illustrer le concept d'héritage:

- Vehicule
- Animal

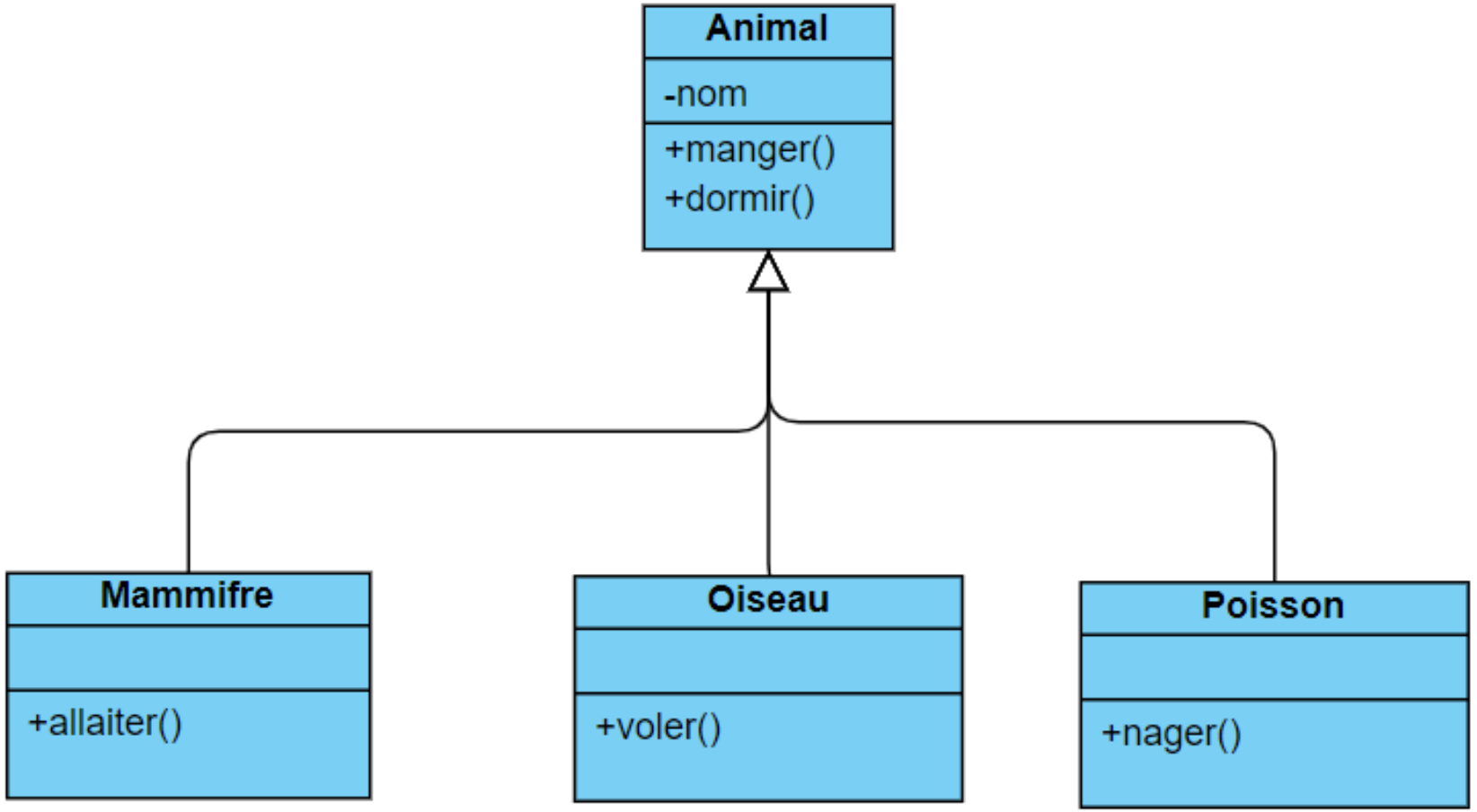
L'Héritage

Exemples :



L'Héritage

Exemples :



Polymorphisme

Définition:

Polymorphisme: signifie « plusieurs formes ». En programmation orientée objet, il se manifeste par la capacité d'une seule interface (ou méthode) à s'adapter à différents types d'objets et à produire des comportements différents en fonction de leur type réel.

Plusieurs types :

- Polymorphisme par sous-typage (ou par héritage)
- Polymorphisme paramétrique
- Polymorphisme ad-hoc

Polymorphisme

Polymorphisme par héritage

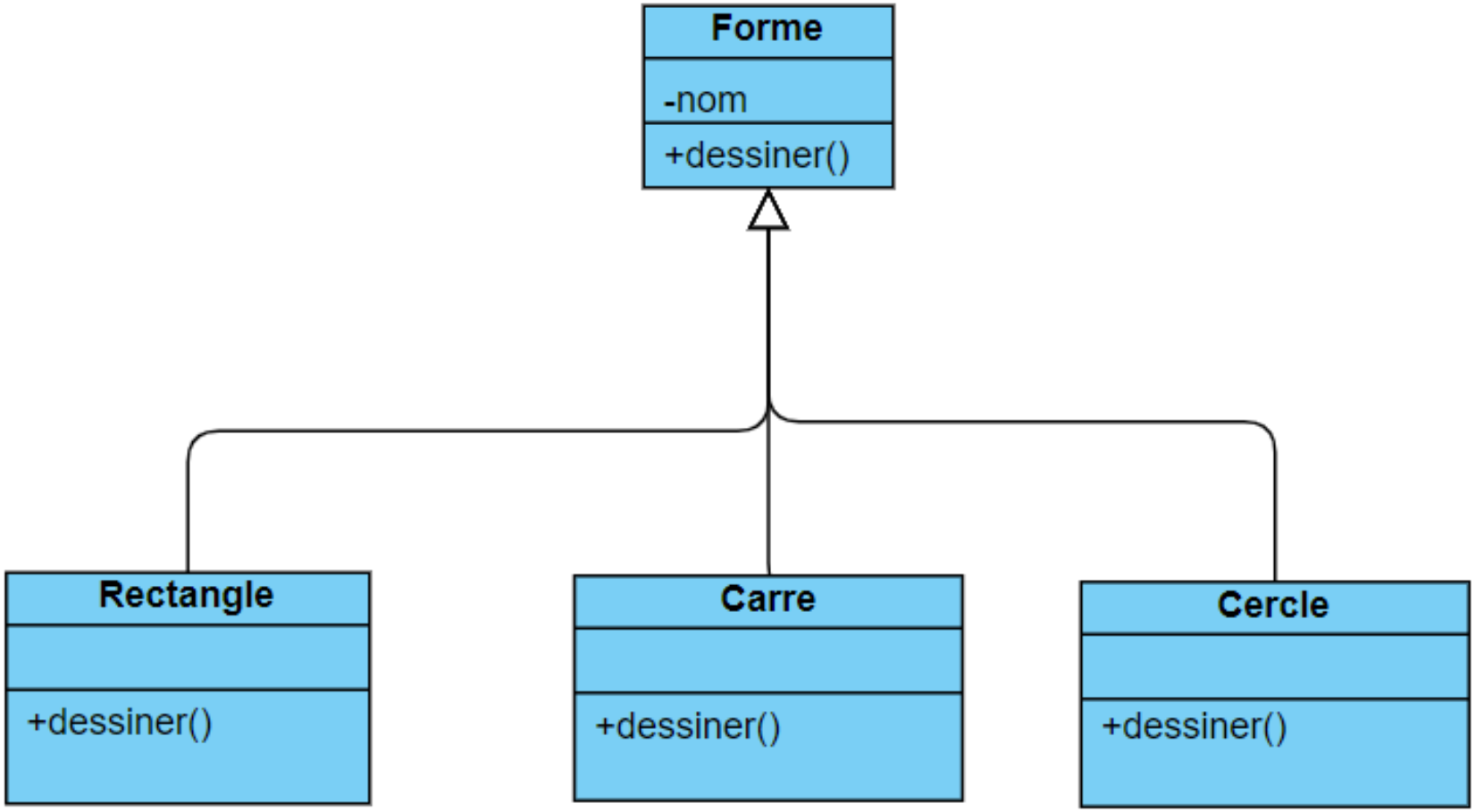
Définition :

Le polymorphisme par héritage est un concept fondamental de la programmation orientée objet (POO) où une classe dérivée (ou sous-classe) peut remplacer ou étendre les comportements définis dans une classe de base (ou super-classe). Ce type de polymorphisme permet d'utiliser des objets de sous-classes à travers une référence de super-classe, offrant une flexibilité et une extensibilité accrues.

Polymorphisme

Polymorphisme par héritage

Exemple:



Polymorphisme

Polymorphisme Ad-Hoc

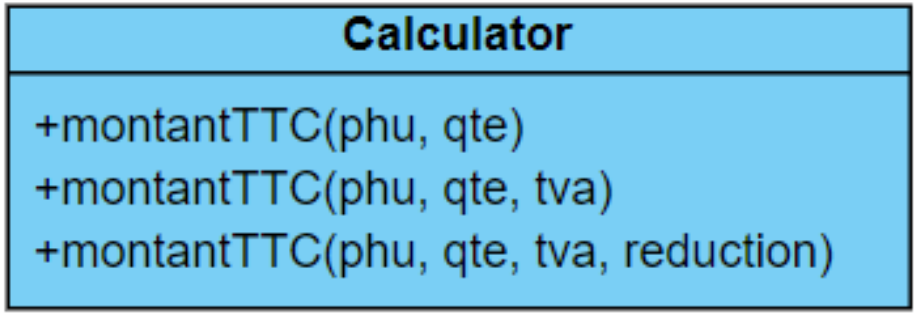
Définition :

Le polymorphisme ad-hoc est un type de polymorphisme qui permet d'utiliser une même fonction ou méthode avec différents types de données, souvent en fonction de la signature (nombre ou type d'arguments) des appels.

Polymorphisme

Polymorphisme Ad-Hoc

Exemples :



Mise en œuvre pratique de la POO au travers d'un langage de programmation :

La POO en Python

Notions de base

Définir et implémenter une classe :

```
class NomDeClass:

    # Constructeur de la classe
    def __init__(self, **args):
        self.attribut_1 = valeur_1
        self.attribut_2 = valeur_2
        ...
        self.attribut_n = valeur_n

    ...
```

Notions de base

Définir et implémenter une classe :

```
class Chat:

    # Constructeur de la classe
    def __init__(self):
        self.race   = None
        self.name   = None
        self.hungry = 0
        self.energy = 0
        self.mood   = 0
```

Notions de base

Définir et implémenter une classe :

```
class Voiture:

    # Constructeur de la classe
    def __init__(self):
        self.marque     = None
        self.annee      = None
        self.carburant  = None
        self.vitesse    = None
        self.niveau_carburant    = None
        self.transmission        = None
```

Notions de base

Instancier une classe :

```
nom_objet = NomDeClass( paramètres )
nom_objet.nom_attribut = valeur
nom_objet.nom_methode( paramètres )
```

Notions de base

Instancier une classe

```
c1 = Chat()
c1.race = 'Siamois'
c1.name = 'Kati'
C1.hungry = 8
c1.energy = 2
c1.mood = 3

c2 = Chat()
c2.race = 'Persan'
c2.name = 'Mika'
C2.hungry = 2
c2.energy = 9
c2.mood = 8
```

Notions de base

Définir et implémenter une classe :

```
class Chat:

    # Constructeur de la classe
    def __init__(self, r, n, h, e, m):
        self.race   = r
        self.name   = n
        self.hungry = h
        self.energy = e
        self.mood   = m
```

Notions de base

Définir et implémenter une classe :

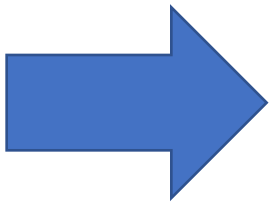
```
class Chat:

    # Constructeur de la classe
    def __init__(self, race, name, hungry, energy, mood):
        self.race   = race
        self.name   = name
        self.hungry = hungry
        self.energy = energy
        self.mood   = mood
```

Notions de base

Instancier une classe

```
c1 = Chat()
c1.race = 'Siamois'
c1.name = 'Kati'
C1.hungry = 8
c1.energy = 2
c1.mood = 3
```



```
c1 = Chat('Siamois', 'Kati', 8, 2, 3)
```

```
c2 = Chat()
c2.race = 'Persan'
c2.name = 'Mika'
C2.hungry = 2
c2.energy = 9
c2.mood = 8
```

```
c2 = Chat('Persan', 'Mika', 2, 9, 8)
```

Notions de base

Définir et implémenter une classe :

```
class Chat:
      # Constructeur de la classe
      def __init__(self, r, n, h, e, m):
            self.race   = r
            self.name   = n
            self.hungry = h
            self.energy = e
            self.mood   = m

      # Méthodes
      def sleep(self):
            self.energy += 1
            self.mood += 1
            self.hungry -= 1
```

Notions de base

Définir et implémenter une classe :

```
class Chat:
        # Constructeur de la classe
        def __init__(self, r, n, h, e, m):
                self.race   = r
                self.name   = n
                self.hungry = h
                self.energy = e
                self.mood   = m

        # Méthodes
        def seep(self):
                self.energy += 1
                self.mood += 1
                self.hungry -= 1

        def feed(self):
                self.hungry -= 1
                self.mood += 1
                self.meow()
```

Notions de base

Instancier une classe

```
c1 = Chat('Siamois', 'Kati', 8, 2, 3)
c1.play()
c1.feed()


c2 = Chat('Persan', 'Mika', 2, 9, 8)
c2.sleep()
c2.meow()
c2.feed()
```

Notions de base

Définir et implémenter une classe (Méthode avec paramètres):

```
class Chat:
      # Constructeur de la classe
      def __init__(self, r, n, h, e, m):
            self.race   = r
            self.name   = n
            self.hungry = h
            self.energy = e
            self.mood   = m

      # Méthodes
      def feed(self, p=1):
            self.hungry -= p
            self.mood += p
            self.meow()
```

Notions de base

Instancier une classe

```
c1 = Chat('Siamois', 'Kati', 8, 2, 3)
c1.play()
c1.feed()


c2 = Chat('Persan', 'Mika', 2, 9, 8)
c2.sleep()
c2.meow()
c2.feed(6)
```

FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

DÉPARTEMENT DE MATHÉMATIQUES ET INFORMATIQUE

Introduction au module numpy

Master : MNSA

Pr: Youssef Ouassit

Le module numpy

Introduction

NumPy est une bibliothèque fondamentale pour le calcul scientifique en Python. Elle fournit un support pour les tableaux (arrays) multidimensionnels, ainsi que des fonctions mathématiques et logiques pour opérer sur ces tableaux de manière efficace.

NumPy est particulièrement utilisé pour des tâches nécessitant des opérations sur des données numériques, comme la manipulation de matrices, l'algèbre linéaire, le calcul statistique, et bien plus encore.

Le module numpy

Introduction

NumPy fournit un moyen pratique et efficace de gérer de grandes quantités de données. Il est également très pratique pour effectuer des multiplications matricielles et pour remodeler (reshape) les données. NumPy est rapide, ce qui le rend adapté au traitement de grands ensembles de données.

- NumPy permet d'effectuer des calculs orientés tableaux (arrays).
- Il implémente efficacement les tableaux multidimensionnels.
- Il permet d'effectuer des calculs scientifiques.
- Il est capable d'effectuer la transformée de Fourier et de remodeler les données stockées dans des tableaux multidimensionnels.
- NumPy fournit des fonctions intégrées pour l'algèbre linéaire et la génération de nombres aléatoires.

Aujourd'hui, NumPy, combiné à SciPy et Matplotlib, constitue un ensemble d'outils largement utilisé pour le calcul scientifique avec Python et peut, pour de nombreux usages, servir d'alternative à MATLAB.

Le module numpy

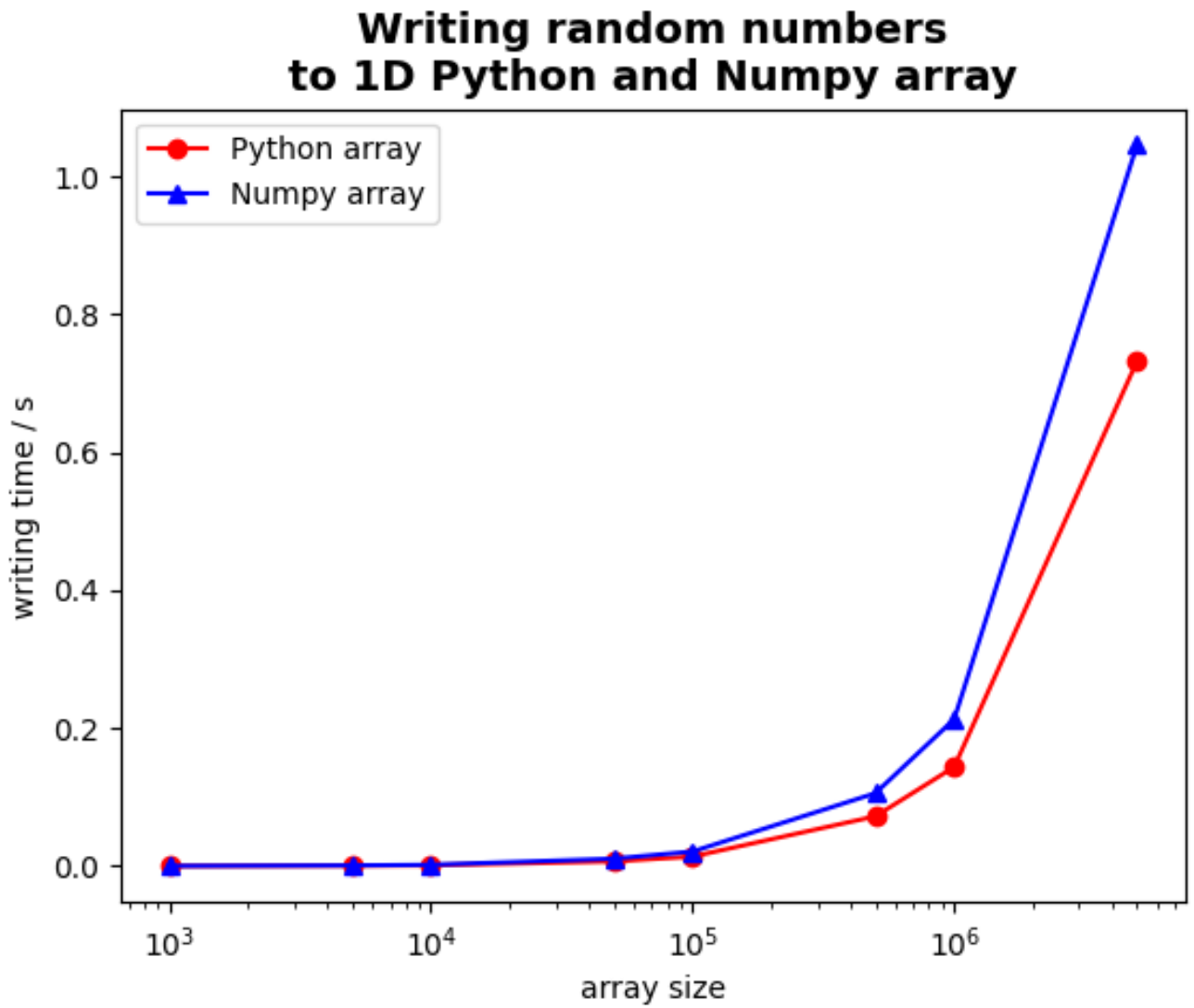
Introduction

Caractéristiques :

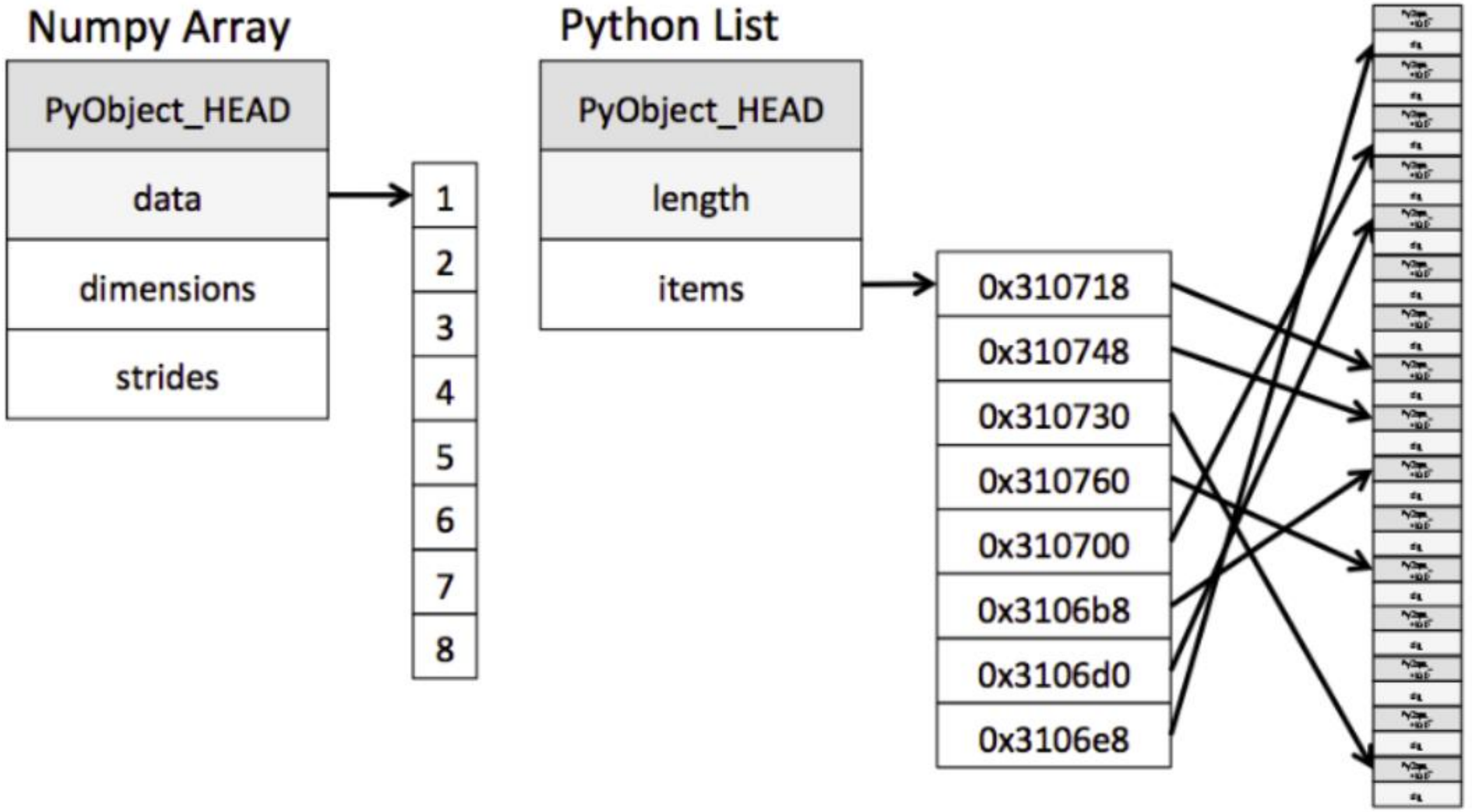
- Tableaux Multidimensionnels (**ndarray**) :
- Opérations Vectorisées (diffusion de l'opérateur)
- Fonctions Mathématiques et Statistiques
- Manipulation de Tableaux
- Génération de Données
- Indexation Avancée et Slicing
- Compatibilité avec d'autres bibliothèques

Le module numpy

Introduction



Introduction



Le module numpy

Importer le module numpy

Pour installer numpy , il faut exécuter la commande suivante :

```
pip install numpy
```

Cette commande est à exécuter sur le console d'exécution ou l'invite de commande.

Le module numpy

Installer le module numpy

Une fois que **NumPy** est installé, vous pouvez l'importer dans votre code Python de la manière suivante :

```
import numpy
```

Pour appeler une fonction :

```
numpy.sin(x)
```

Installer le module numpy

Une fois que **NumPy** est installé, vous pouvez l'importer dans votre code Python de la manière suivante :

```
import numpy as np
```

np est un alias couramment utilisé pour NumPy. Cela vous permet d'utiliser des fonctions et des objets de NumPy de manière plus concise.

Pour appeler une fonction :
np.sin(x)

Le module numpy

Déclaration d'un vecteur

Déclarer un tableau un à une seule dimension

Le module numpy

Déclaration d'un vecteur

- Créer un tableau 1D à partir d'une liste:

La fonction np.array() est l'une des façons les plus courantes de créer des tableaux 1D. Elle convertit une liste Python (ou un autre objet itérable) en un tableau NumPy.

```
A = np.array([1, 2, 3, 4, 5])
```

Le module numpy

Déclaration d'un vecteur

- Créer un tableau 1D rempli de zéros:

La fonction np.zeros() crée un tableau 1D rempli de zéros. Vous devez spécifier la taille du tableau.

```
A = np.zeros(5)      # donne [ 0.0  0.0  0.0  0.0  0.0]
```

Déclaration d'un vecteur

- Créer un tableau 1D rempli des uns:

La fonction np.zeros() crée un tableau 1D rempli des uns. Vous devez spécifier la taille du tableau.

```
A = np.ones(5)     # donne [ 1.0  1.0  1.0  1.0  1.0]
```

Le module numpy

Déclaration d'un vecteur

- Créer un tableau 1D avec une séquence de nombres:

np.arange() génère un tableau avec une séquence d'entiers ou de nombres à intervalles réguliers, similaire à la fonction range() en Python.

```
# Créer un tableau 1D avec une séquence de nombres de 0 à 9
A = np.arange(10)

# Créer un tableau 1D avec des nombres allant de 0 à 10 (pas de 2)
B = np.arange(0, 10, 2)
```

Le module numpy

Déclaration d'un vecteur

- Créer un tableau 1D avec un nombre spécifié d'éléments uniformément espacés:

np.linspace() génère un tableau 1D avec des nombres répartis de manière uniforme entre deux bornes.

```
# Créer un tableau 1D avec 5 nombres uniformément espacés entre 0 et 1
A = np.linspace(0, 1, 5)
```

Déclaration d'un vecteur

- Créer un tableau 1D avec des nombres aléatoires entre 0 et 1:

np.random.rand() crée un tableau 1D avec des nombres aléatoires uniformément distribués entre 0 et 1.

```
# Créer un tableau 1D de 5 éléments avec des valeurs aléatoires entre 0 et 1
A = np.random.rand(5)
```

Déclaration d'un vecteur

- Créer un tableau 1D avec des entiers aléatoires dans une plage donnée:

np.random.randint() génère un tableau 1D d'entiers aléatoires dans une plage spécifiée..

```
# Créer un tableau 1D de 5 entiers aléatoires entre 0 et 10
A = np.random.randint(0, 10, 5)
```

Déclaration d'un vecteur

- Répéter un tableau 1D:

np.tile() répète un tableau 1D plusieurs fois pour créer un tableau plus grand.

```
# Créer un tableau 1D
A = np.array([1, 2, 3])

# Répéter ce tableau 3 fois
B = np.tile(A, 3)  # Donne [1 2 3 1 2 3 1 2 3]
```

Déclaration des matrices

Déclaration des matrices

Déclaration des matrices

- Créer une matrice à partir d'une liste:

La fonction array() crée une matrice de la forme spécifiée, et tous ses éléments sont initialisés à zéro.

```
# Créer une matrice 2x3
M = np.array( [[1,2,3],[4,5,6]] )

# Créer une matrice 2x3
A = np.array( [[1,2,3],[4,5,6], [0,1,2]] )
```

Déclaration des matrices

- Créer une matrice remplie de zéros:

La fonction np.zeros() crée une matrice de la forme spécifiée, et tous ses éléments sont initialisés à zéro.

```
# Créer une matrice 3x3 remplie de zéros
M = np.zeros((3, 3))
```

Déclaration des matrices

- Créer une matrice remplie des uns:

La fonction np.ones() crée une matrice de la forme spécifiée, et tous ses éléments sont initialisés à un.

```
# Créer une matrice 3x3 remplie des uns
M = np.ones((3, 3))
```

Déclaration des matrices

- Créer une matrice identité:

La fonction np.eye() crée une matrice carrée de taille spécifiée, avec des 1 sur la diagonale principale et des 0 ailleurs.

```
# Créer une matrice identité 4x4
M = np.eye(4)
```

Déclaration des matrices

- Créer une matrice diagonale:

np.diag() permet de créer une matrice diagonale à partir d'un tableau 1D. Les éléments du tableau 1D seront placés sur la diagonale principale de la matrice.

```
# Créer une matrice diagonale à partir d'un tableau
M = np.diag([1, 2, 3])
```

Déclaration des matrices

- Créer une matrice avec des nombres aléatoires uniformément répartis:

np.random.rand() génère une matrice remplie de nombres aléatoires entre 0 et 1, suivant une distribution uniforme.

```
# Créer une matrice 3x3 avec des nombres aléatoires entre 0 et 1
M = np.random.rand(3, 3)
```

Déclaration des matrices

- Créer une matrice avec des entiers aléatoires:

np.random.randint() génère une matrice avec des entiers aléatoires dans une plage spécifiée.

```
# Créer une matrice 3x3 avec des entiers aléatoires entre 0 et 10
M= np.random.randint(0, 10 , (3, 3))
```

Le module numpy

Déclaration d'un vecteur

Fonctions

Déclaration d'un vecteur

On retrouve les mêmes fonctions déjà vus :

- **len**
- **max**
- **min**
- **sum**
- **sorted**

Changer la dimension et la forme d'un tableau

La fonction reshape de NumPy permet de changer la forme (dimensions) d'un tableau sans modifier ses données. Elle prend en entrée un tableau NumPy et une nouvelle forme, et retourne un tableau avec la nouvelle forme spécifiée.

```
import numpy as np

np.reshape(A, newshape)
```

Exemple:

```
A = np.arange(1, 10)
A = np.reshape(A, (3, 3))
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Déclaration d'un vecteur

Indexation et extraction des valeurs

Indexation d'un vecteur

L'indexation nous permet d'accéder aux éléments de la liste. Cependant, il existe d'autres moyens, tels que l'indexation de fantaisie, le découpage et le masquage.

```
A = np.array([1, 2, 3, 4])

# Accéder au dernier  élément de A
#  Ces  deux  instructions  d'impression  génèrent  le  même
résultat
print(A[-1])
print(A[3])
```

Indexation d'un vecteur

```
A = np.array([7, 0, 1, 9, 8, 5])

#plage d'indices contigus
print(A[1:3]) # [0  1]

#extrêmes, début à 3 (non-inclus)
print(A[:3]) # [7  0  1]

#extrêmes, 2 à fin
print(v[2:]) # [1  9  8  5]
```

Indexation d'un vecteur

```
A = np.array([7, 0, 1, 9, 8, 5])

#indice négatif
print(A[-1]) # 5  , dernier élément

#indices négatifs
print(A[-3:]) # [9  8  5], 3 derniers éléments

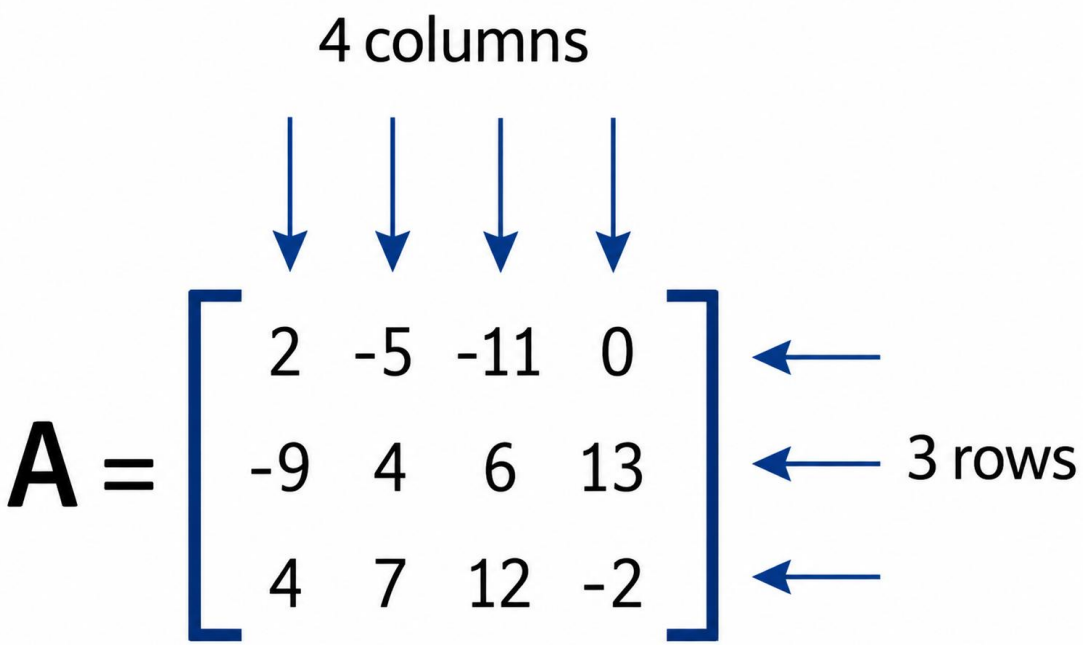
#on peut utiliser une condition pour l'extraction
A[ A < 7] # [0  1  5]
```

Indexation d'une matrice

```
# Accéder à une valeur

A[0][0]  # donne 2
A[1][1]  # donne 4
A[2][3]  # donne -2

A[0 , 0]  # donne 2
A[1 , 1]  # donne 4
A[2 , 3]  # donne -2
```



Indexation d'une matrice

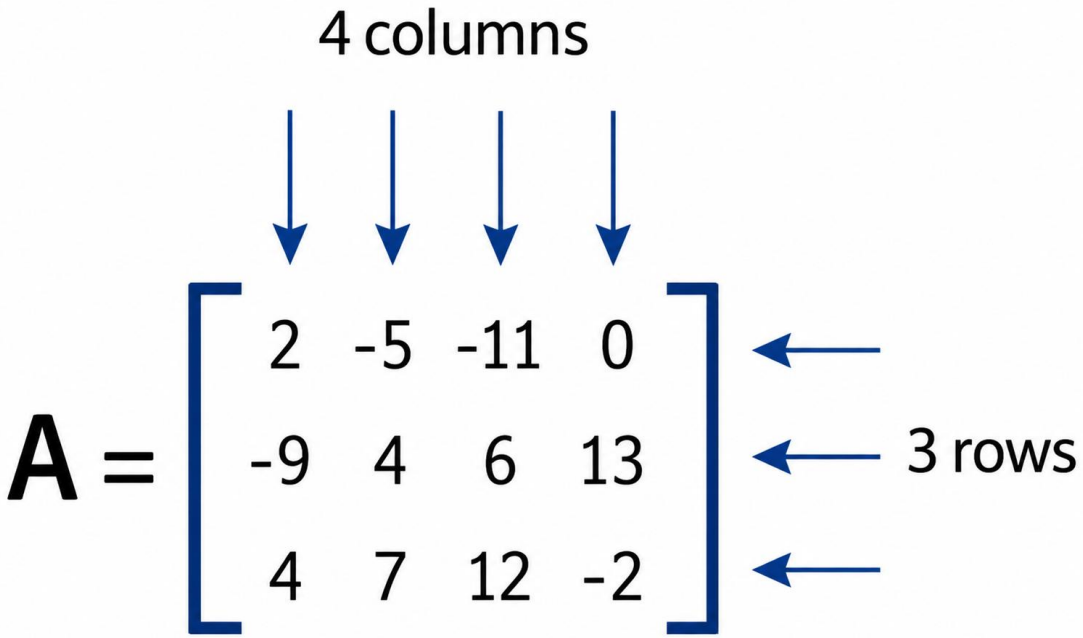
```
# Accéder à une valeur
```

```
A[ 1: , 2: ]
```

$$\begin{bmatrix} 6 & 13 \\ 12 & -2 \end{bmatrix}$$

```
A[ 1:3 , 1:3 ]
```

$$\begin{bmatrix} 4 & 6 \\ 7 & 12 \end{bmatrix}$$



Indexation d'une matrice

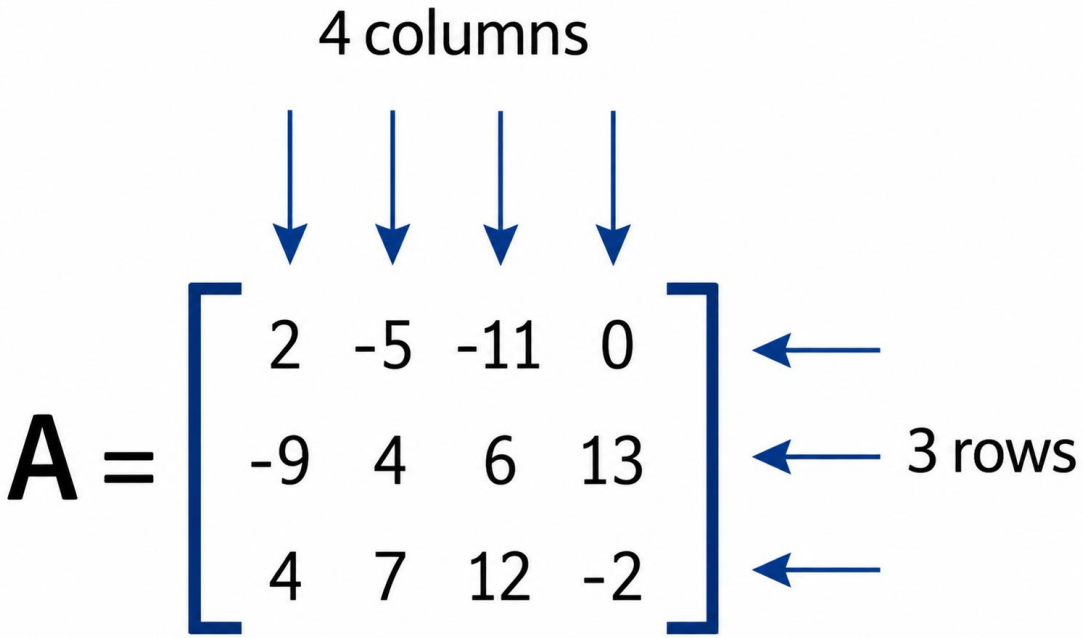
```
# Accéder à une valeur
```

```
A[ 1: , 2: ]
```

$$\begin{bmatrix} 6 & 13 \\ 12 & -2 \end{bmatrix}$$

```
A[ 1:3 , 1:3 ]
```

$$\begin{bmatrix} 4 & 6 \\ 7 & 12 \end{bmatrix}$$



Indexation d'une matrice

Exemples :

a[:3, :5]

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

a[-3:, -3:]

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

Indexation d'une matrice

Exemples :

a[:, 3]

0	1	2	**3**	4	5	6	7
8	9	10	**11**	12	13	14	15
16	17	18	**19**	20	21	22	23
24	25	26	**27**	28	29	30	31
32	33	34	**35**	36	37	38	39

a[1, 3:6]

0	1	2	3	4	5	6	7
8	9	10	**11**	**12**	**13**	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39

Indexation d'une matrice

Exemples :

a[1::2, ::3]

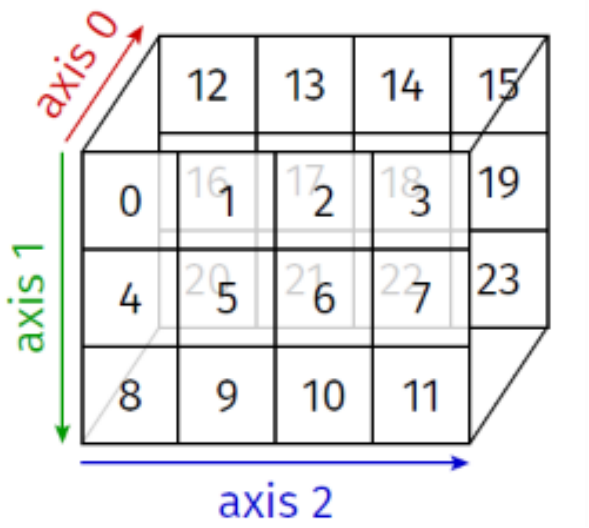
0	1	2	3	4	5	6	7
8	9	10	**11**	12	13	**14**	15
16	17	18	19	20	21	22	23
24	25	26	**27**	28	29	**30**	31
32	33	34	35	36	37	38	39

a[a%3==3]

0	1	2	**3**	4	5	**6**	7
8	**9**	10	11	**12**	13	14	**15**
16	17	**18**	19	20	**21**	22	23
24	25	26	**27**	28	29	**30**	31
32	**33**	34	35	**36**	37	38	**39**

Indexation d'une matrice

Exemples :



Déclaration d'un vecteur

La diffusion de l'opérateur

La diffusion de l'opérateur

La **diffusion** (ou **broadcasting**) est une fonctionnalité puissante de **NumPy** qui permet d’effectuer des opérations sur des tableaux de différentes formes (dimensions) sans avoir besoin de les rendre explicitement de même forme.
Cela signifie que **NumPy** peut automatiquement étendre (ou "diffuser") un tableau plus petit pour qu’il corresponde à la forme d'un tableau plus grand, tout en respectant les règles de diffusion.

La diffusion de l'opérateur

Addition d'un scalaire à un tableau:

```
# Un tableau 1D
A = np.array([1, 2, 3])

# Ajouter un scalaire
B  = A + 10   # 10 est diffusé sur chaque élément du tableau  donne [11 12 13]
```

La diffusion de l'opérateur

Multiplication d'un scalaire à un tableau:

```
# Un tableau 1D
A = np.array([1, 2, 3])

# Ajouter un scalaire
B  = A * 10   # 10 est diffusé sur chaque élément du tableau  donne
[10  20  30]
```

La diffusion de l'opérateur

Diffusion entre deux tableaux de formes compatibles:

```
# Un tableau 2D
A = np.array([1, 2, 3])

# Un tableau 1D
B = np.array([10, 20, 30])

# Addition de A à B
C = A + B # A est diffusé sur chaque ligne de B, donne [  11  22  33 ]
```

La diffusion de l'opérateur

Diffusion entre deux tableaux de formes compatibles:

```
# Un tableau 2D
A = np.array([1, 2, 3])

# Un tableau 1D
B = np.array([10, 20, 30])

# Multiplication de A à B
C = A * B # A est diffusé sur chaque ligne de B, donne [  10  40  90 ]
```

La diffusion de l'opérateur

Appliquer des fonctions:

```
# Un tableau 2D
A = np.array([4, 16, 36])

# Racine carré de A
np.sqrt(A) # donne [  2   4   6]
```

La diffusion de l'opérateur

Appliquer des fonctions:

```
# Un tableau 2D
A = np.array([4, 16, 36])

# Que donne ces instructions
R = np.sin(A)**2  + np.cos(A)**2  # [ 1.0  1.0  1.0]
```

Les fonctions statistiques

Les fonctions statistiques

Les fonctions statistiques

Calculer la moyenne d'un tableau:

```
A = np.array([1, 2, 3, 4, 10])
m = np.mean(A)  # donne 4
```

Les fonctions statistiques

Calculer la médiane d'un tableau:

```
A = np.array([1, 2, 3, 4, 6])
m = np.median(A)  # donne 3

A = np.array([1, 2, 3, 4])
m = np.median(A)  # donne 2.5
```

Les fonctions statistiques

Calculer l'écart-type d'un tableau:

```
A = np.array([1, 2, 3, 4])
m = np.std(A)  # donne 1.118
```

Les fonctions statistiques

Trouver l'index du maximum ou du minimum:

```
A = np.array([1, 7, 0, 4])
max_index = np.argmax(A)  # 1 (indice du plus grand élément)
min_index = np.argmin(A)  # 2 (indice du plus petit élément)
```

Opérations sur les matrices

Opérations sur les matrices

Opérations sur les matrices

Somme matricielle :

$$\begin{pmatrix} 1 & 2 \\ 1 & 3 \end{pmatrix} + \begin{pmatrix} 4 & 1 \\ 5 & 3 \end{pmatrix} = \begin{pmatrix} 5 & 3 \\ 6 & 6 \end{pmatrix}$$

```
A = np.array([ [1,2], [1,3]] )
B = np.array([ [4,1], [5,3]] )

S = A + B
```

Opérations sur les matrices

produit matriciel :

$$\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 1 & -1 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} 9 \\ 2 \end{bmatrix}$$

```
A = np.array([ [2,3], [1,-1]] )
B = np.array([ [9], [2]] )

P = np.dot(A , B)
```

Opérations sur les matrices

Déterminant d'une matrice :

```
A = np.array([ [2,3], [1,-1]] )

P = np.linalg.det(A)
```

FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

DÉPARTEMENT DE MATHÉMATIQUES ET INFORMATIQUE

Introduction au module matplotlib

Master : MNSA

Pr: Youssef Ouassit

Introduction au module matplotlib

Plan

1. Introduction
2. Tracer de courbes
3. Travailler avec des Sous-Graphiques
4. Visualisation des fonctions à deux variables
5. Visualisation d’une fonction à valeurs complexes avec Python
6. Animation avec Matplotlib
7. Autres bibliothèques : Seaborn, Plotly

Introduction au module matplotlib

Introduction

Matplotlib est une bibliothèque Python puissante et flexible pour la visualisation de données en 2D. Elle permet de créer divers types de graphiques, comme des courbes, des diagrammes en barres, des nuages de points et des histogrammes.

Matplotlib est une bibliothèque de visualisation de données multiplateforme construite sur des tableaux NumPy et conçue pour fonctionner avec l'ensemble plus large de SciPy. Elle a été introduite par John Hunter en 2002.

L'un des plus grands avantages de la visualisation est qu'elle nous permet d'accéder visuellement à de grandes quantités de données sous une forme facilement digestible.

Matplotlib comprend plusieurs types de tracés comme les courbes, les barres, les nuages de points, les histogrammes, etc.

Introduction

Deux styles de programmation :

- le style « pyplot » qui utilise directement des fonctions du module pyplot ;
- le style « Oriénté Objet (OO) » qui est recommandé dans la documentation de Matplotlib.

Introduction

Les modules de la bibliothèque matplotlib:

- matplotlib.pyplot
- matplotlib.figure
- matplotlib.axes
- matplotlib.patches
- matplotlib.lines
- matplotlib.text
- matplotlib.animation
- matplotlib.image
- …

Introduction au module matplotlib

Introduction

Importer les modules :

```
import numpy as np
import matplotlib.pyplot as plt
```

Introduction au module matplotlib

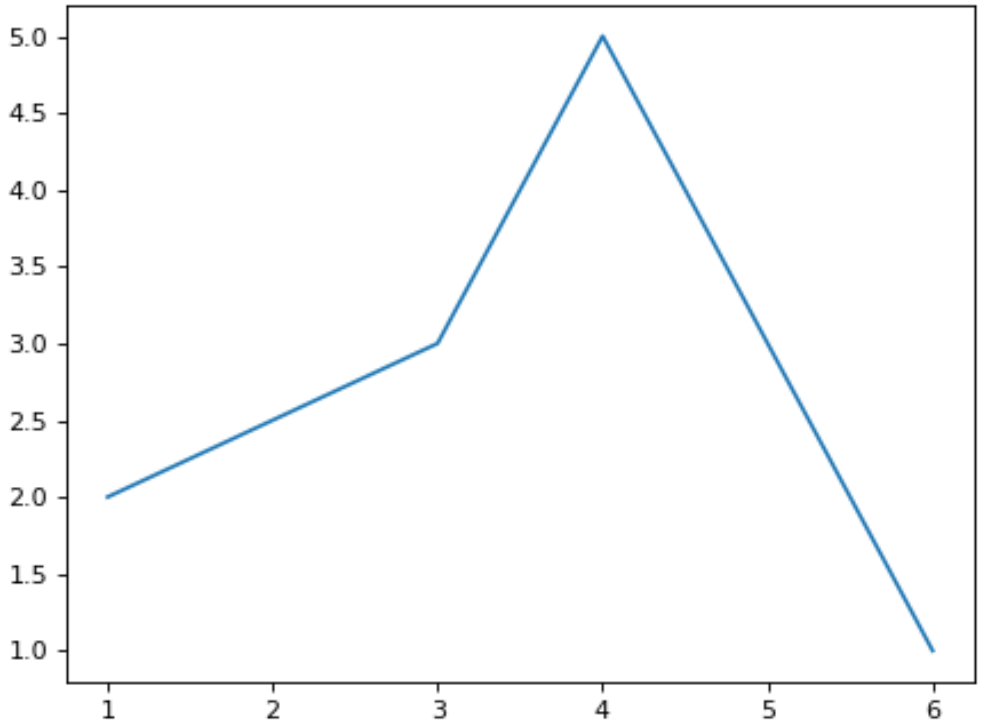
Tracer une courbe simple

Style pyplot :

```
import numpy as np
import matplotlib.pyplot as plt

plt.plot([1, 3, 4, 6], [2, 3, 5, 1])

plt.show()
```



Introduction au module matplotlib

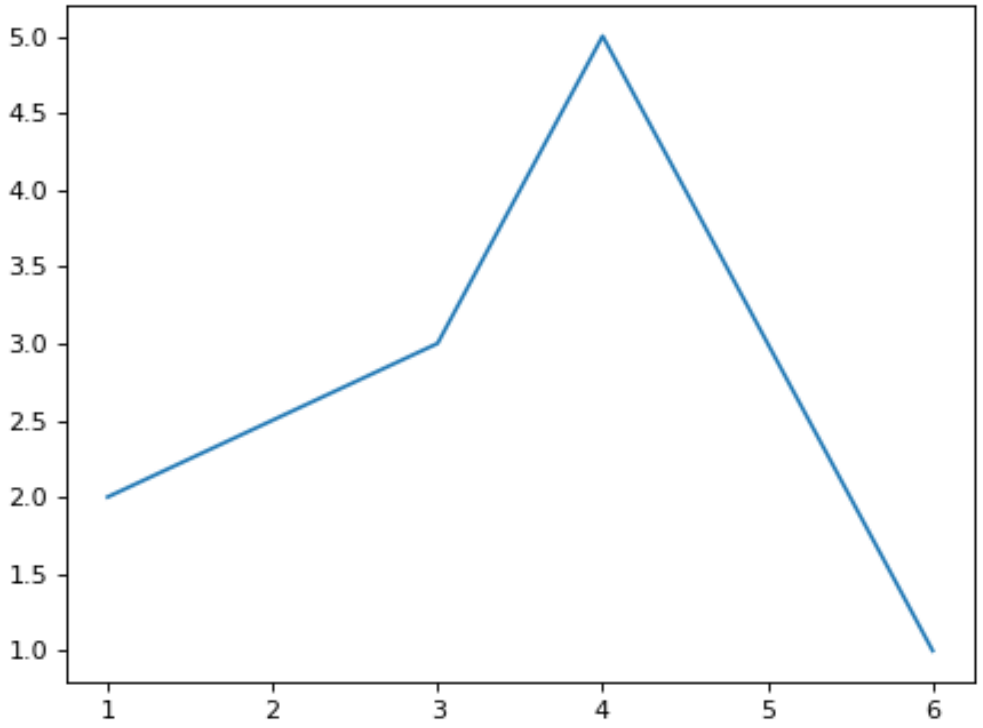
Tracer une courbe simple

Style Orienté Objet :

```
import numpy as np
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot([1, 3, 4, 6], [2, 3, 5, 1])

plt.show()
```



Introduction au module matplotlib

Tracer une courbe simple

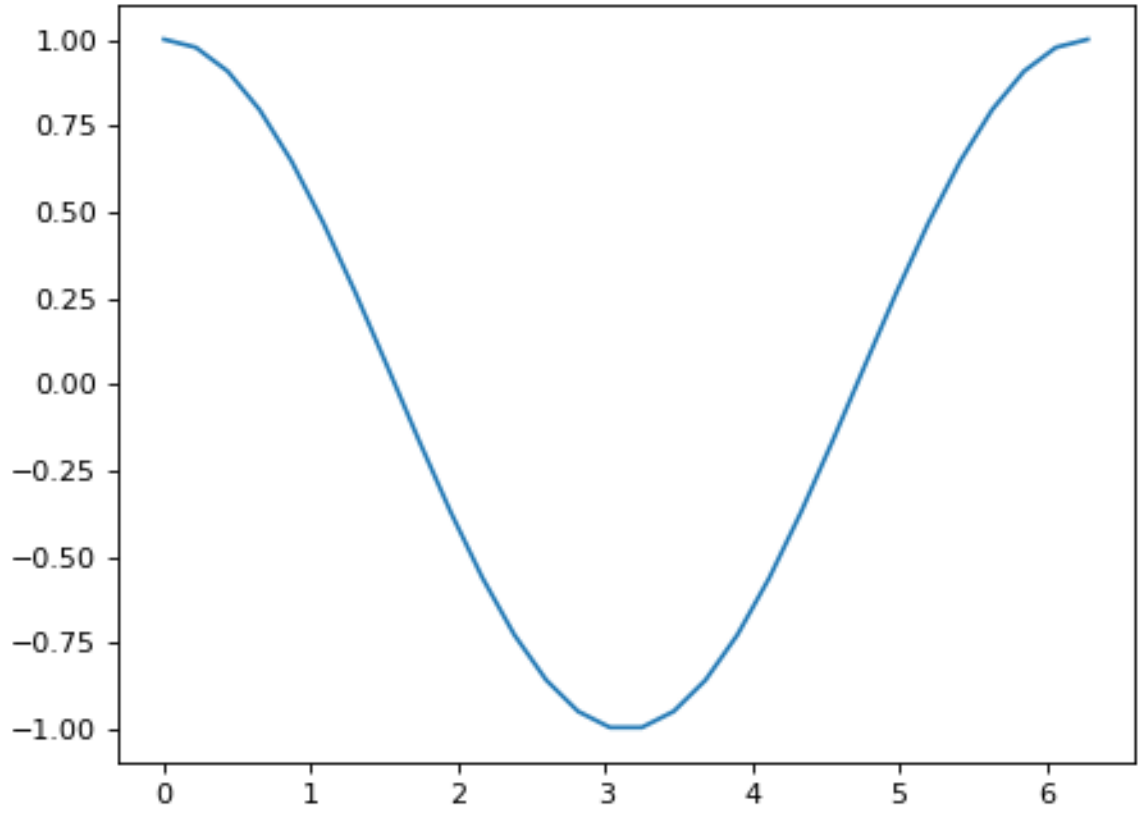
La courbe de la fonction cos :

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 30)
y = np.cos(x)

plt.plot(x, y)

plt.show()
```



Introduction au module matplotlib

Tracer une courbe simple

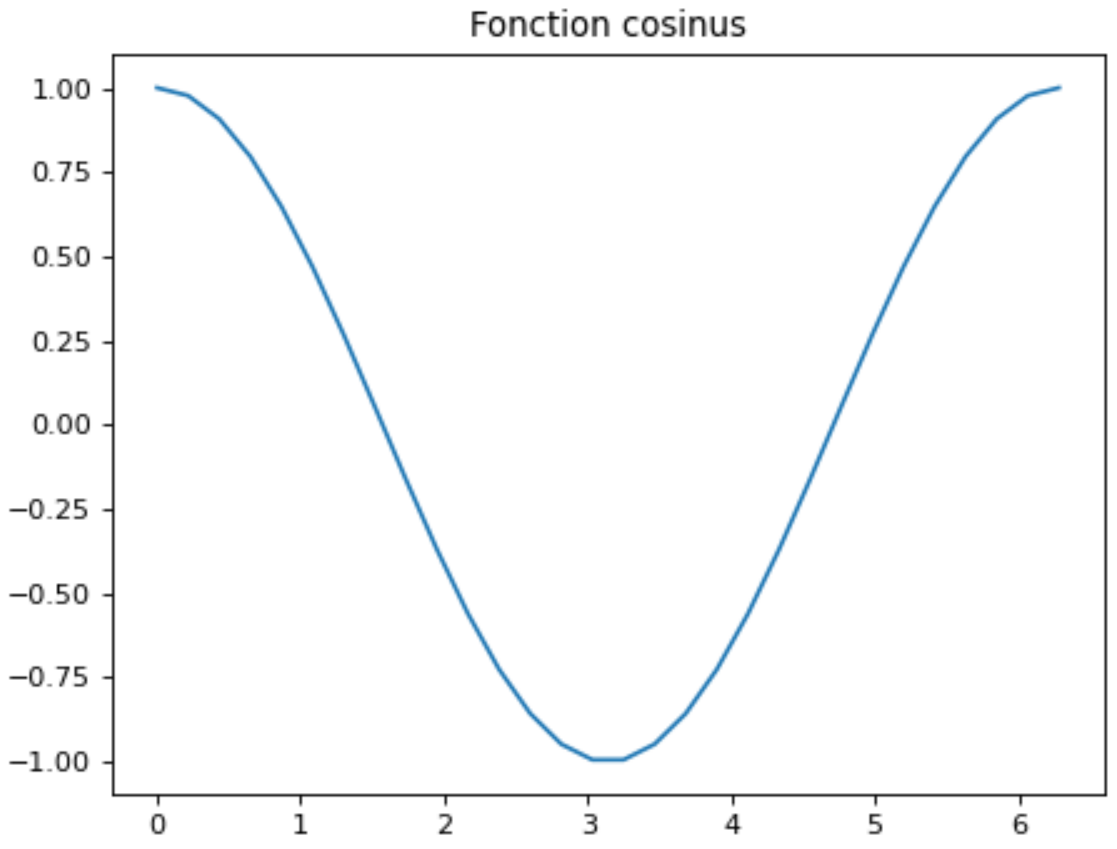
Ajouter un titre (style pyplot):

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 30)
y = np.cos(x)

plt.plot(x, y)
plt.title("Fonction cosinus")

plt.show()
```



Introduction au module matplotlib

Tracer une courbe simple

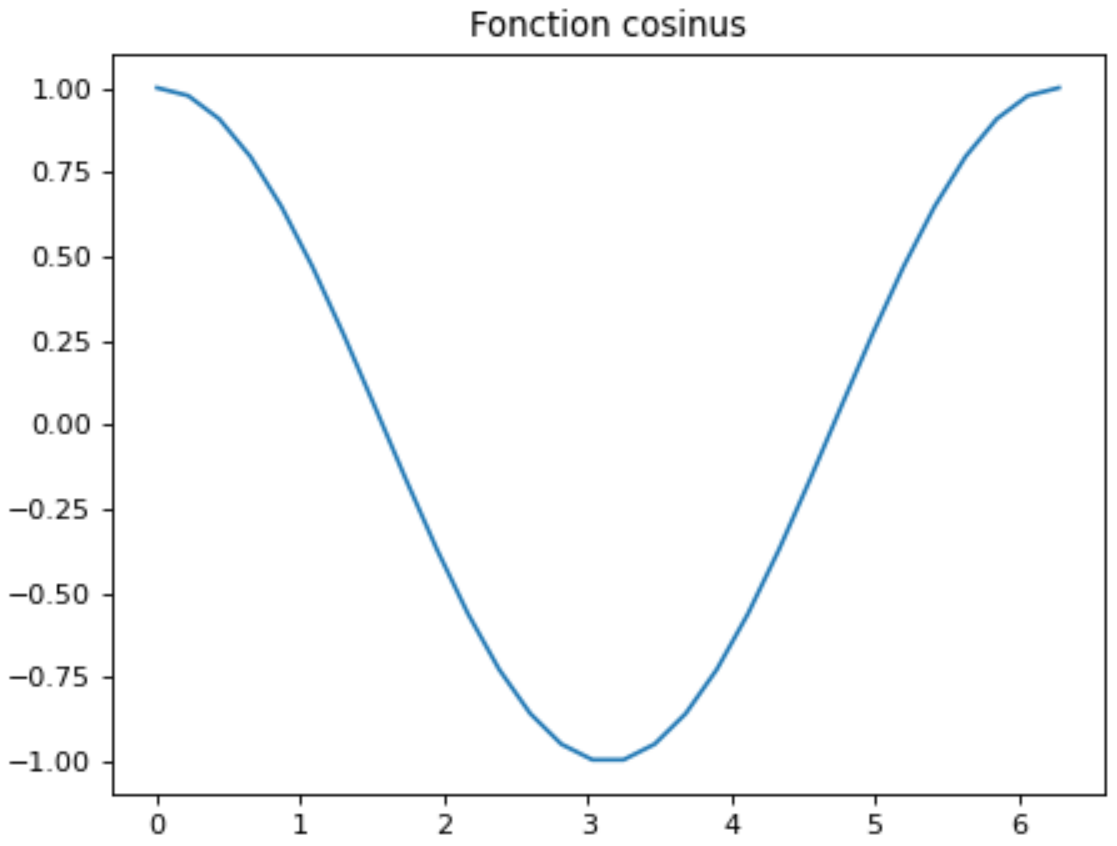
Ajouter un titre (style orienté objet):

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 30)
y = np.cos(x)

fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title("Fonction cosinus")

plt.show()
```



Introduction au module matplotlib

Tracer une courbe simple

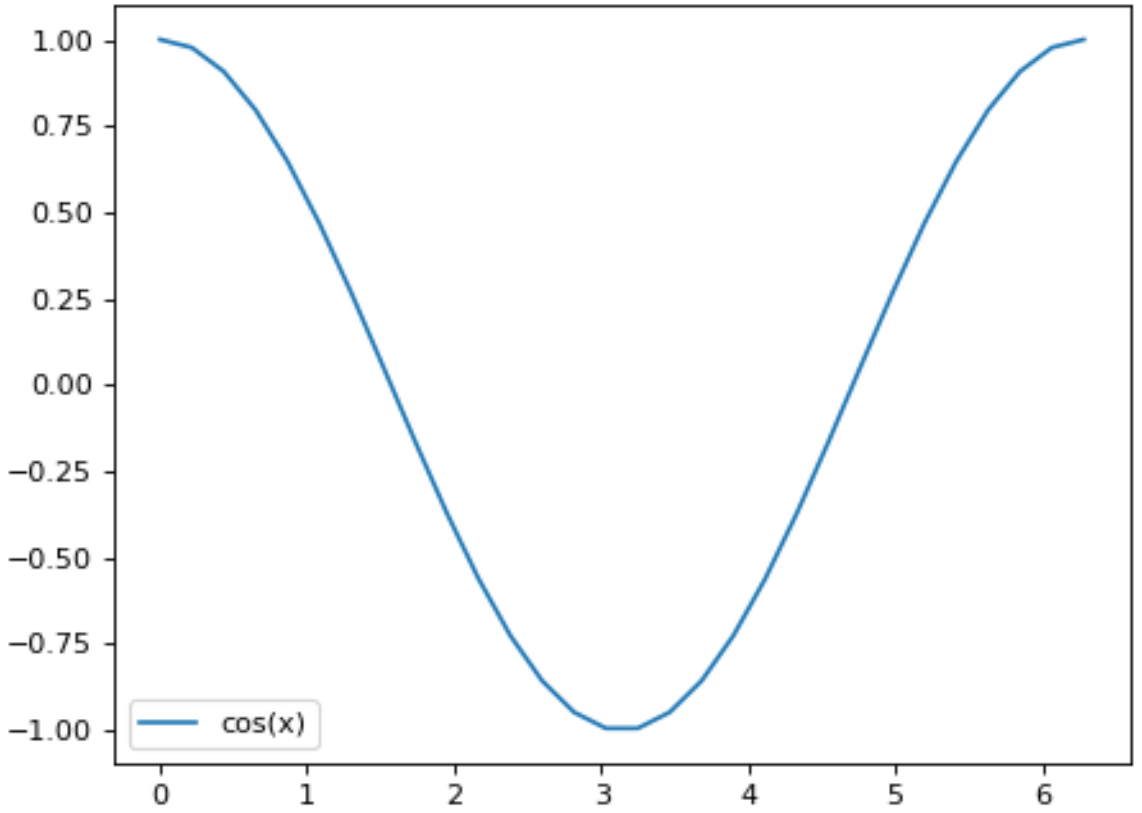
Ajouter une légende:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 30)
y = np.cos(x)

plt.plot(x, y, label="cos(x)")
plt.title("Fonction cosinus")
plt.legend()

plt.show()
```



Introduction au module matplotlib

Tracer une courbe simple

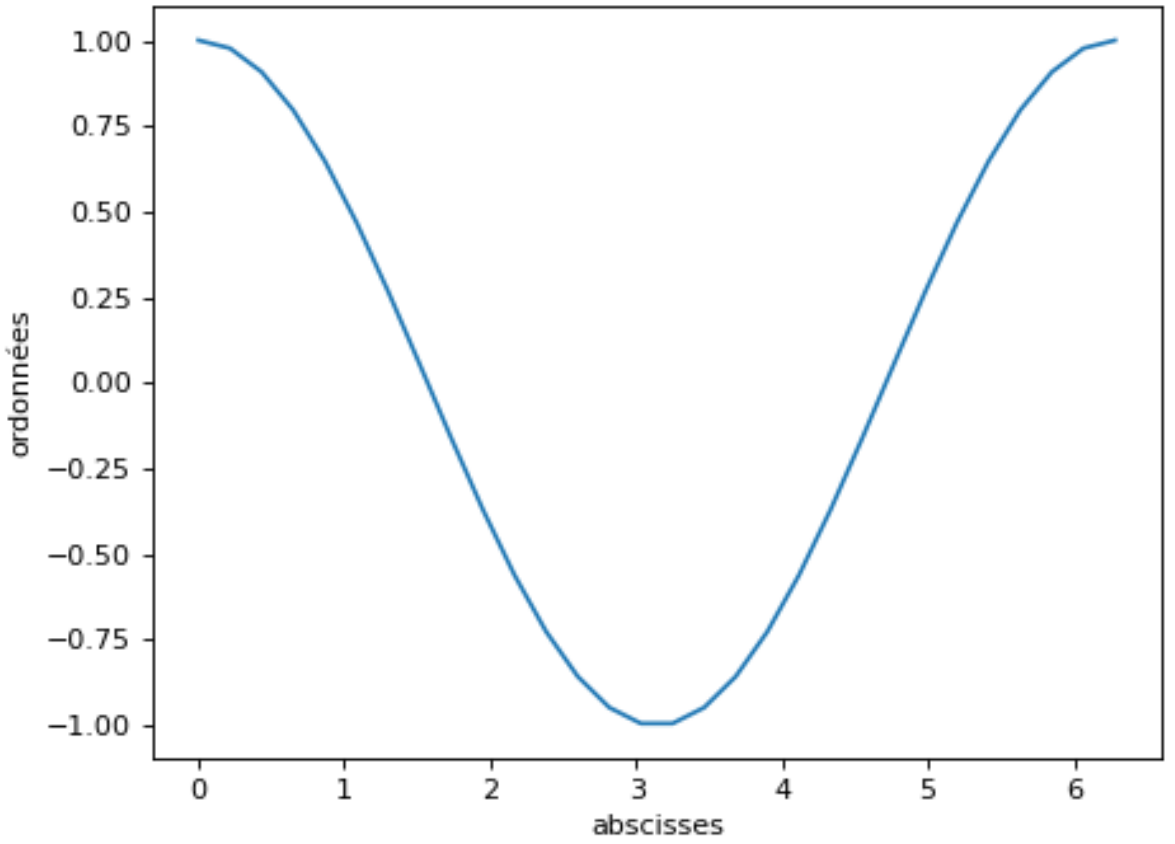
Ajouter des labels sur les axes:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 30)
y = np.cos(x)

plt.plot(x, y)
plt.xlabel("abscisses")
plt.ylabel("ordonnées")

plt.show()
```



Introduction au module matplotlib

Tracer une courbe simple

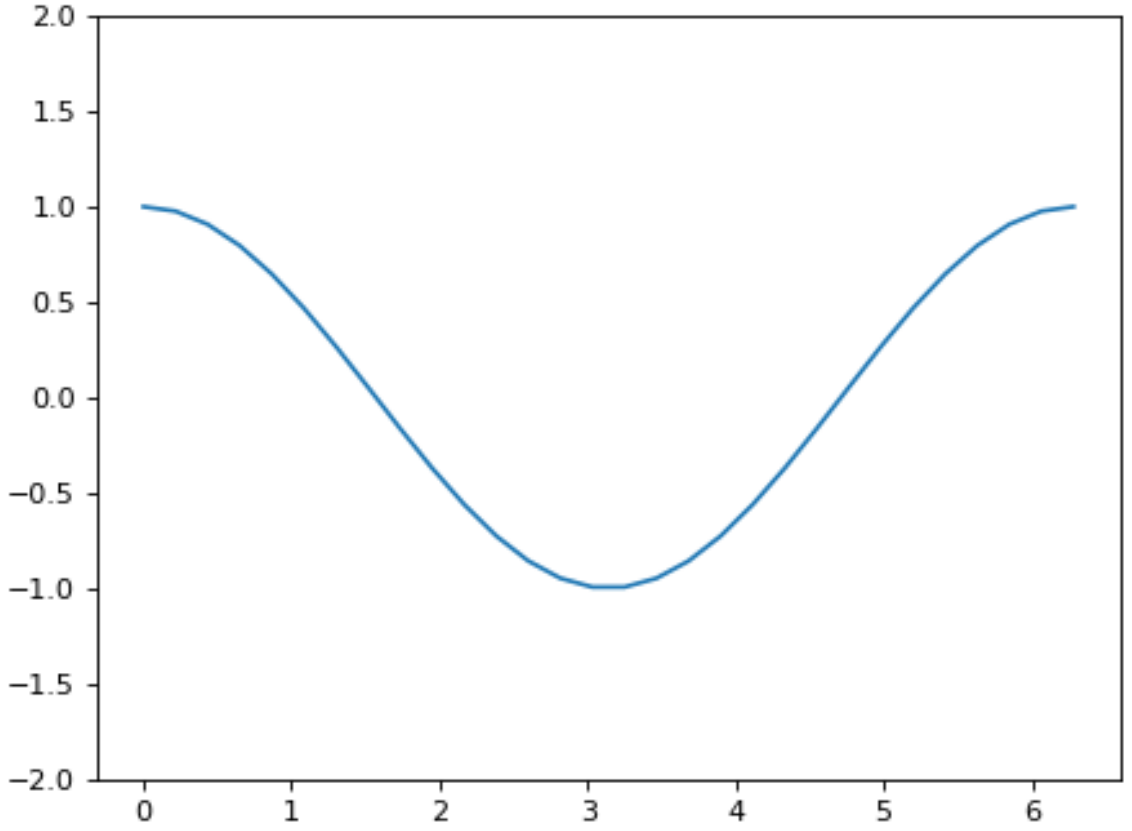
Définition du domaine des axes:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 30)
y = np.cos(x)

plt.plot(x, y)
plt.xdim(-1, 5)
plt.ydim(-2, 2)

plt.show()
```



Introduction au module matplotlib

Tracer une courbe simple

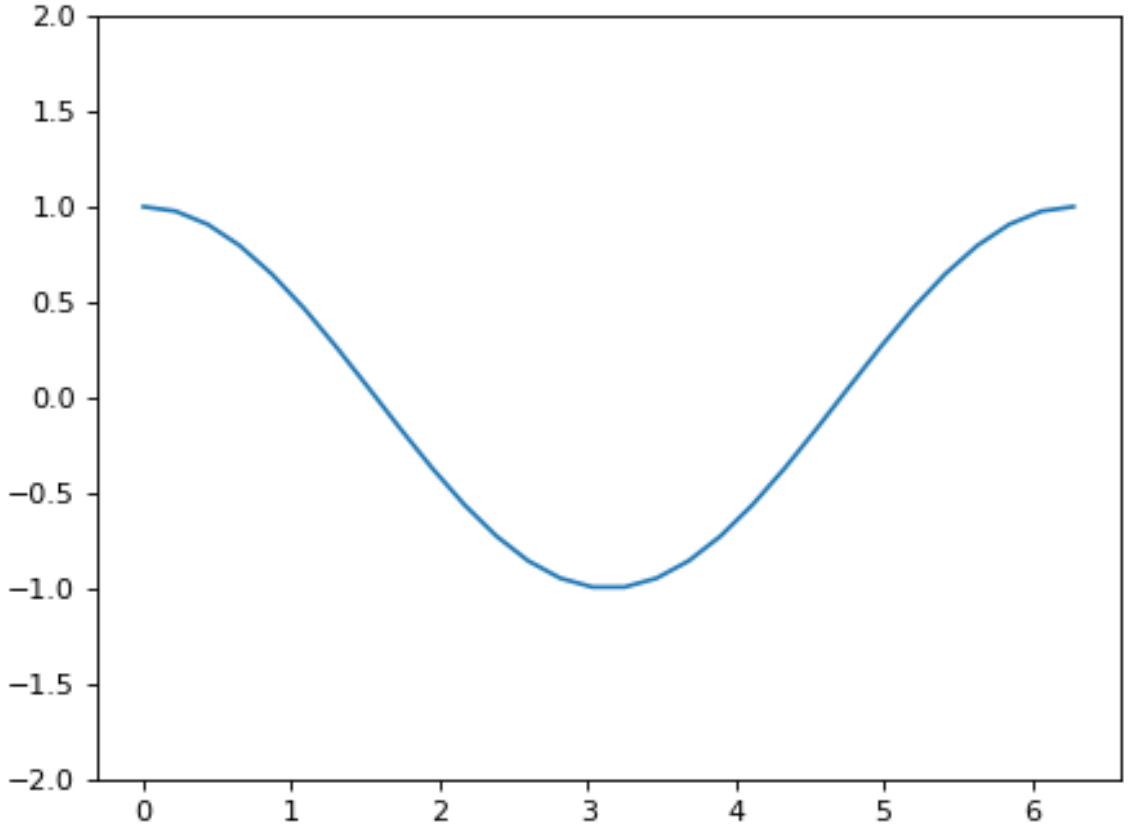
Définition du domaine des axes:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 30)
y = np.cos(x)

plt.plot(x, y)
plt.axis([-1, 5, -2, 2])

plt.show()
```



Introduction au module matplotlib

Tracer une courbe simple

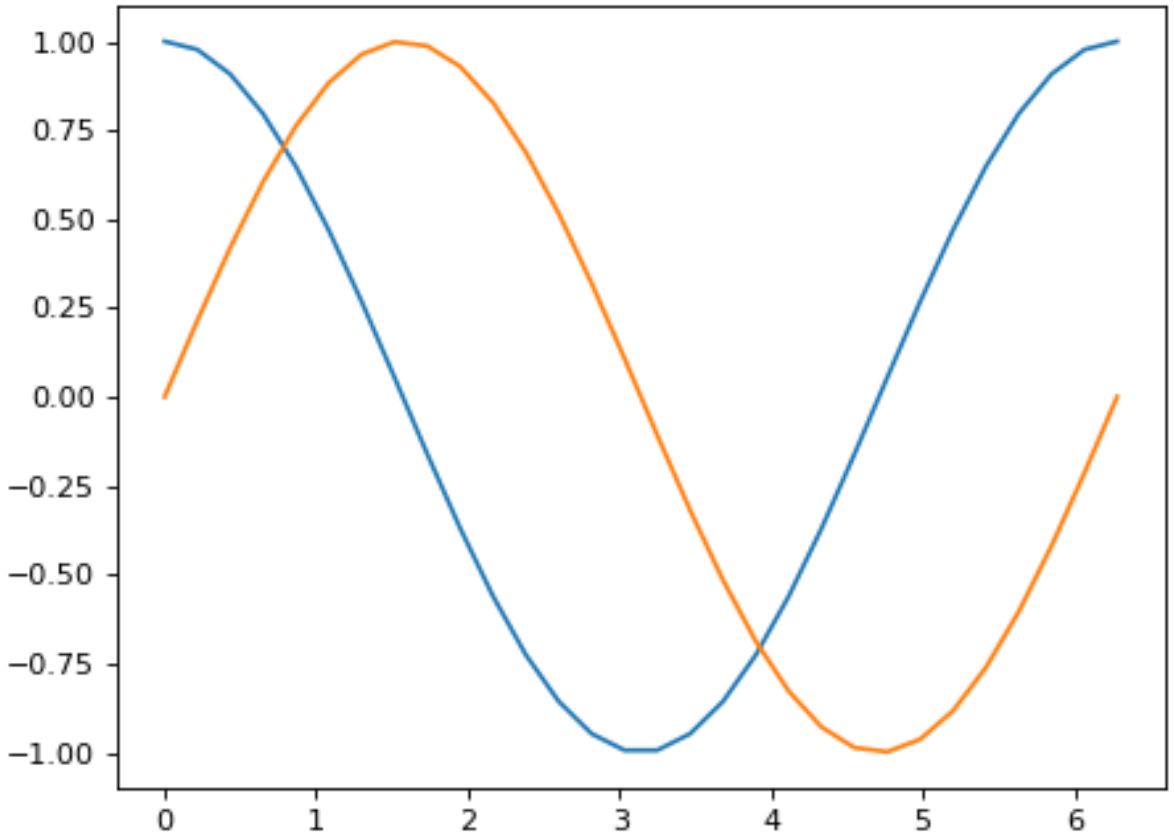
Afficher plusieurs courbes:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 30)
y1 = np.cos(x)
y2 = np.sin(x)

plt.plot(x, y1)
plt.plot(x, y2)

plt.show()
```



Introduction au module matplotlib

Tracer une courbe simple

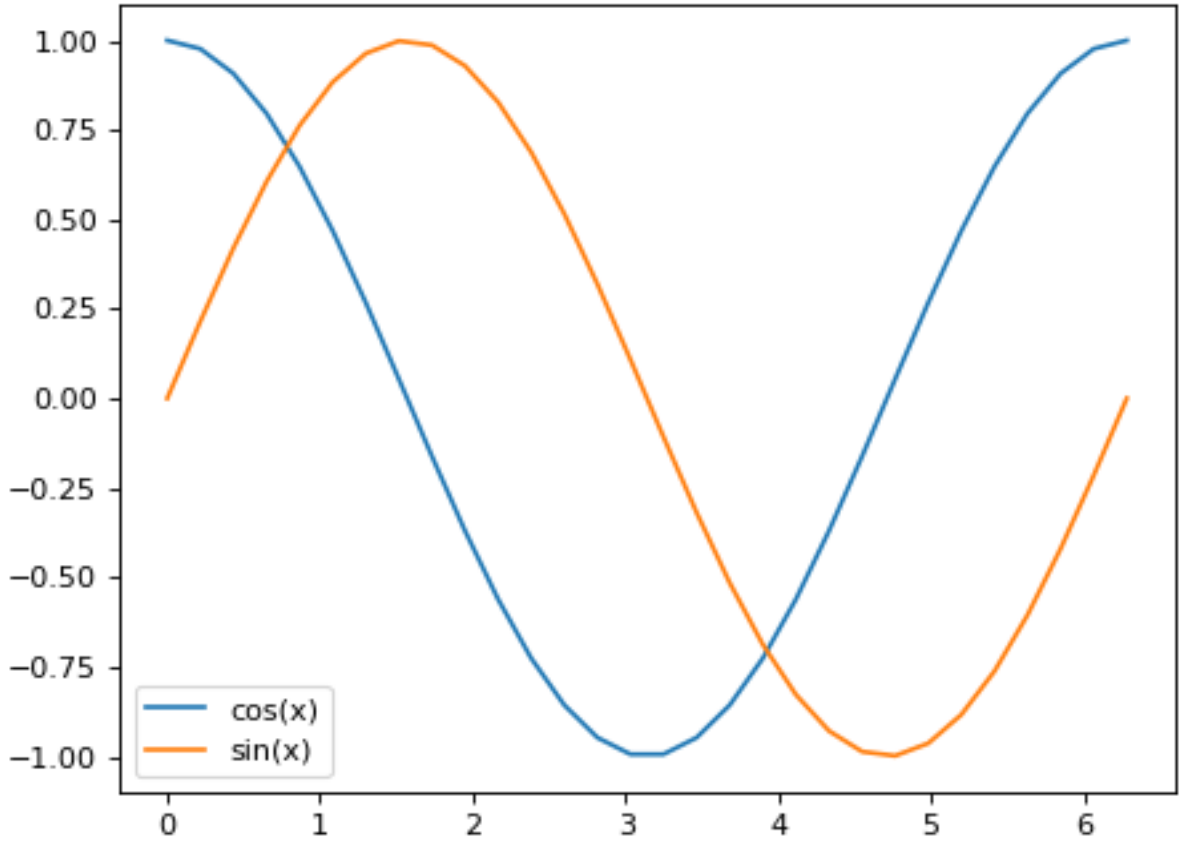
Afficher plusieurs courbes:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 30)
y1 = np.cos(x)
y2 = np.sin(x)

plt.plot(x, y1, label="cos(x)")
plt.plot(x, y2, label="sin(x)")
Plt.legend()

plt.show()
```



Introduction au module matplotlib

Tracer une courbe simple

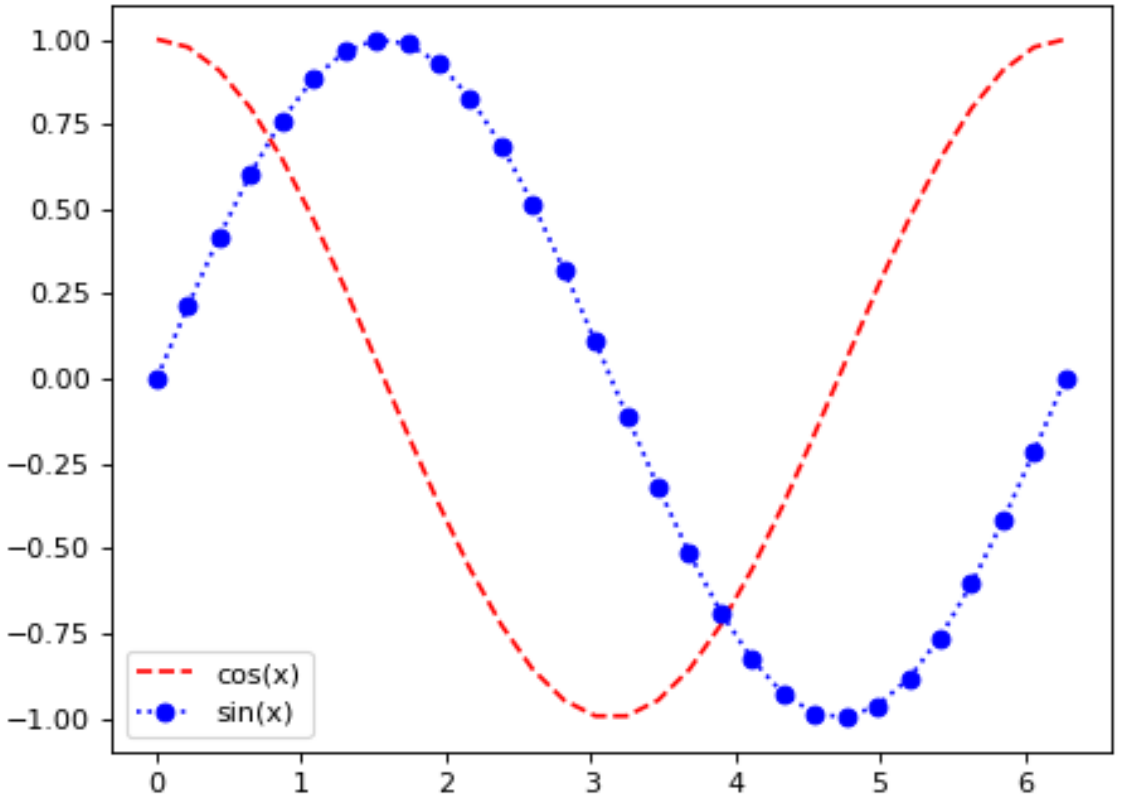
Afficher plusieurs courbes:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 30)
y1 = np.cos(x)
y2 = np.sin(x)

plt.plot(x, y1, "r--", label="cos(x)")
plt.plot(x, y2, "b:o", label="sin(x)")
plt.legend()

plt.show()
```



Tracer une courbe simple

Style de ligne

Les chaînes de caractères suivantes permettent de définir le style de ligne :

Chaîne	Effet
-	Ligne continue
- -	Tirets
:	Ligne en pointillé
-.	Tirets points

Tracer une courbe simple

Symbole (Marker)

Les chaînes de caractères suivantes permettent de définir le symbole (« marker ») :

Chaîne	Effet
.	point marker
,	pixel marker
o	circle marker
v	triangle_down marker
^	triangle_up marker
….	

Tracer une courbe simple

Couleur

Les chaînes de caractères suivantes permettent de définir la couleur :

Chaîne	Effet
b	blue
g	green
r	red
c	cyan
y	yellow
m	magenta
k	black
w	white

Tracer des formes

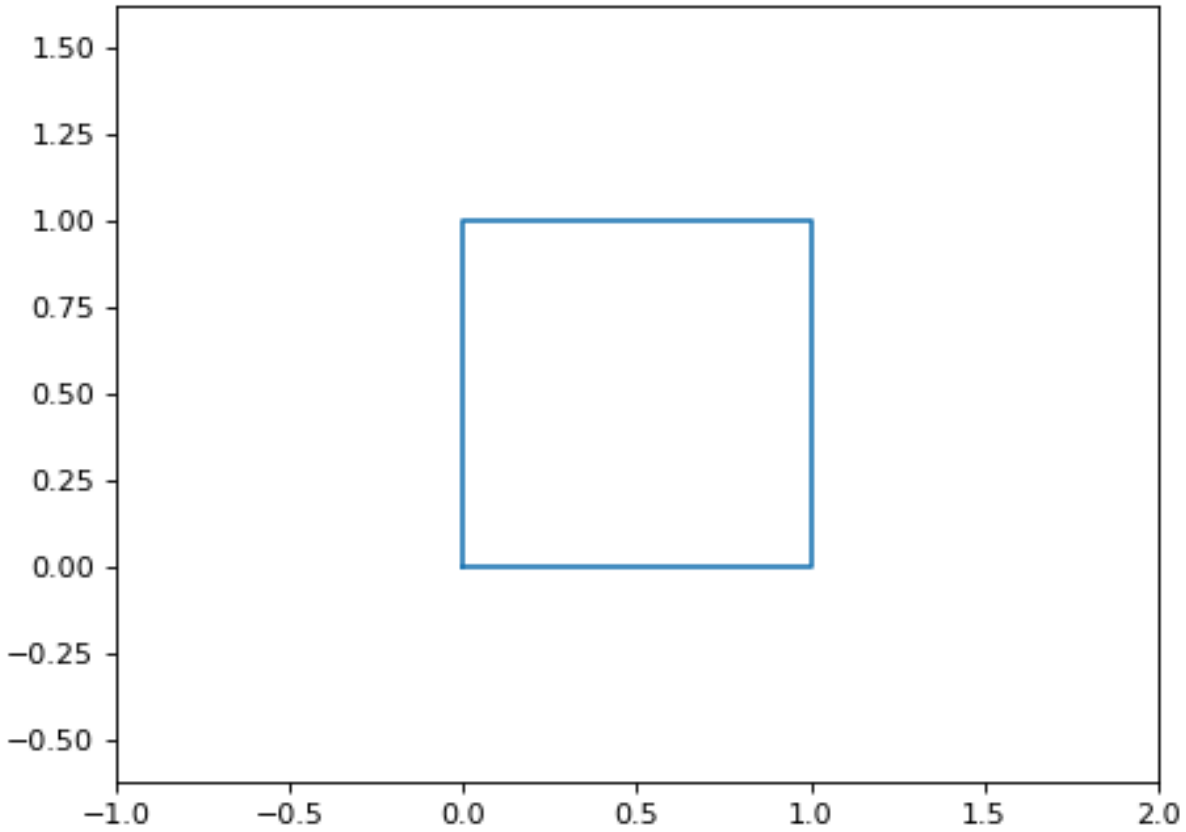
Tracer un rectangle

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([0, 1, 1, 0, 0])
y = np.array([0, 0, 1, 1, 0])

plt.plot(x, y)
plt.axis("equal")
plt.axis([-1, 2, -1, 2])

plt.show()
```



Introduction au module matplotlib

Tracer des formes

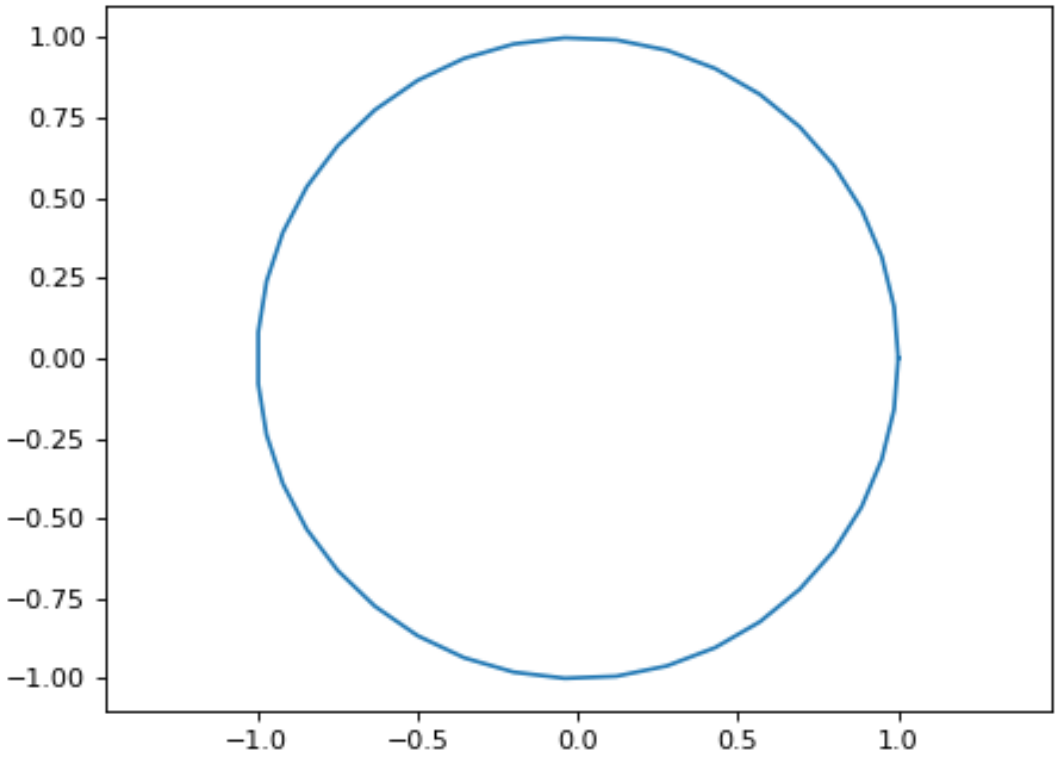
Tracer un cercle

```
import numpy as np
import matplotlib.pyplot as plt

theta = np.linspace(0, 2*np.pi, 40)
r = 1
x = r*np.cos(theta)
y = r*np.sin(theta)

plt.plot(x, y)
plt.axis("equal")

plt.show()
```



Plusieurs tracés sur la même figure

La fonction **plt.subplots(nrows=1, ncols=1)** de Matplotlib est une méthode pratique pour créer plusieurs sous-graphiques (subplots) organisés en une grille dans une seule figure.

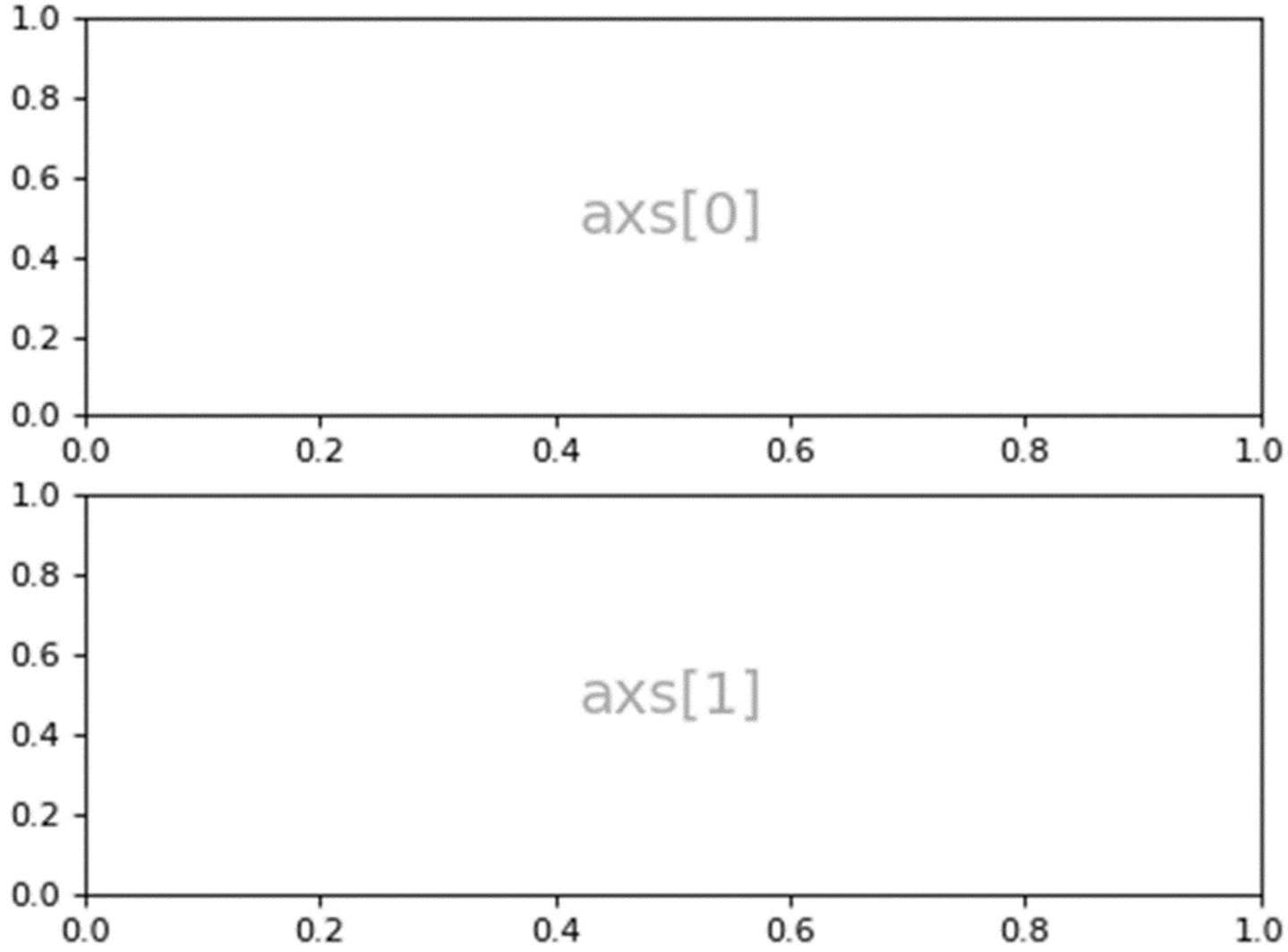
Cela permet de visualiser plusieurs graphiques côte à côte ou dans une disposition en grille, tout en partageant le même espace de figure.

Introduction au module matplotlib

Plusieurs tracés sur la même figure

Exemple:

L'instruction **plt.subplots(2, 1)** permet de créer deux graphiques en deux lignes et une colonne.

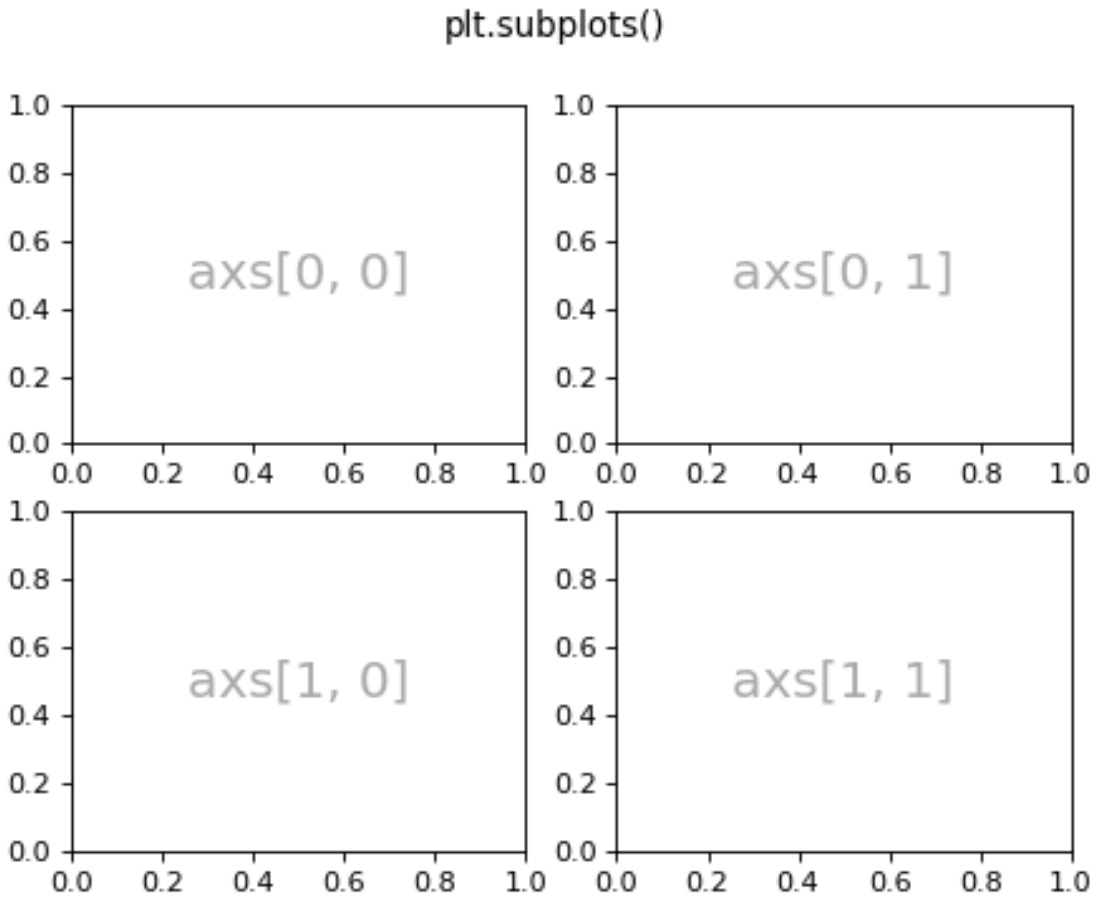


Introduction au module matplotlib

Plusieurs tracés sur la même figure

Exemple:

L'instruction **plt.subplots(2, 2)** permet de créer quatre graphiques en deux lignes et deux colonnes.

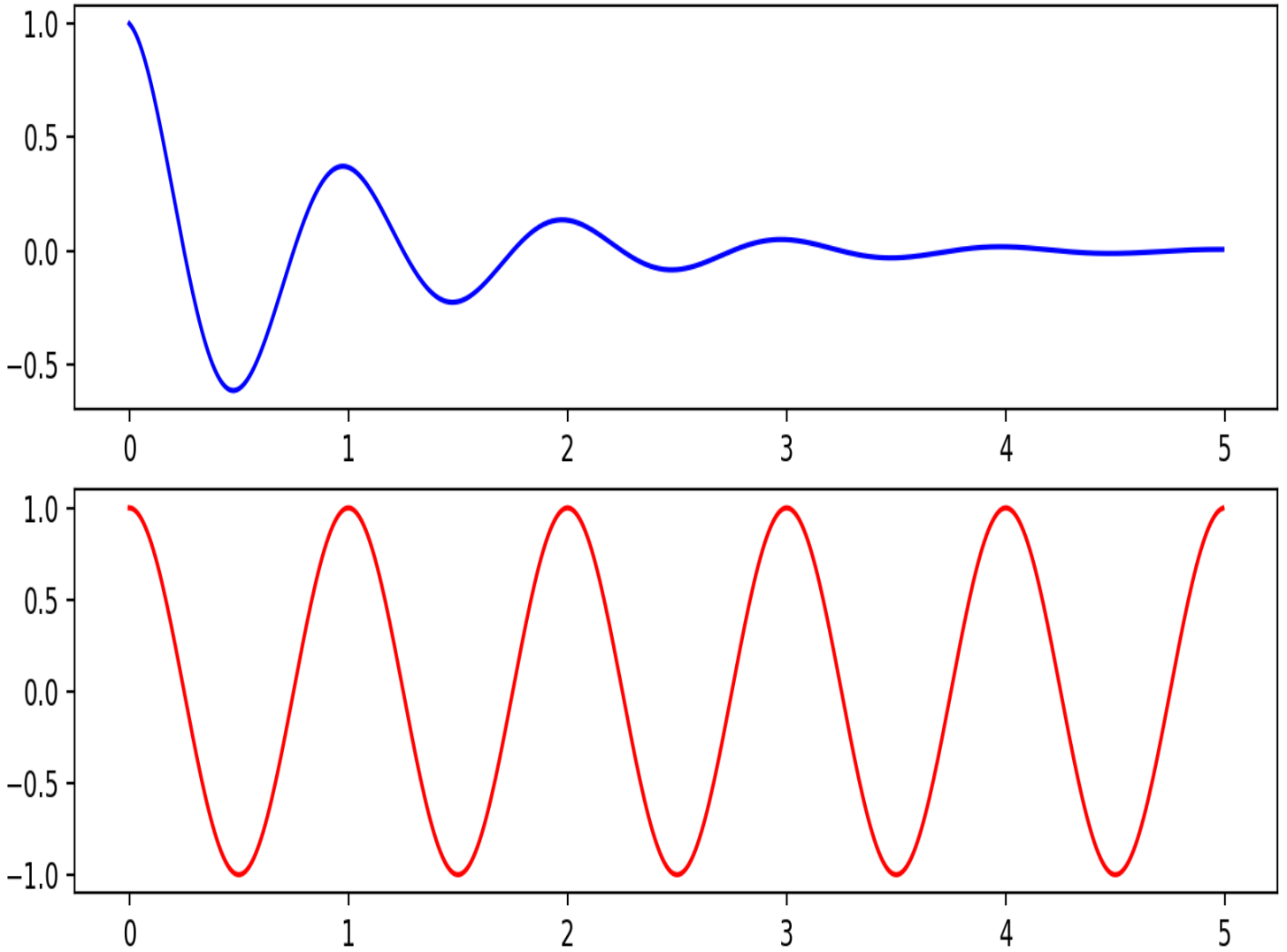


Introduction au module matplotlib

Plusieurs tracés sur la même figure

$$f(x) = e^{-t}\cos(2\pi t)$$

$$\mathrm{g}(x) = \cos(2\pi t)$$



Introduction au module matplotlib

Plusieurs tracés sur la même figure

```
import numpy as np
import matplotlib.pyplot as plt

f = lambda t : np.exp(-t) * np.cos(2*np.pi*t)
g = lambda t : np.cos(2*np.pi*t)

t = np.arange(0.0, 5.0, 0.01)

fig, axs = plt.subplots(2, 1)

axs[0].plot(t, f(t), "b")
axs[1].plot(t, g(t), "r")

plt.show()
```

Introduction au module matplotlib

Visualiser une fonction à deux variables

Pour visualiser une fonction z = f(x, y) :

Il faut d'abord générer des tableaux X et Y (noms arbitraires) qui contiennent les valeurs des abscisses et ordonnées pour chacun des points grâce à la fonction meshgrid().

Ensuite il faut calculer la valeur de z pour chacun de ces points.

Introduction au module matplotlib

Visualiser une fonction à deux variables

Utilisation de pcolormesh() : permet une visualisation grâce à des couleurs.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)

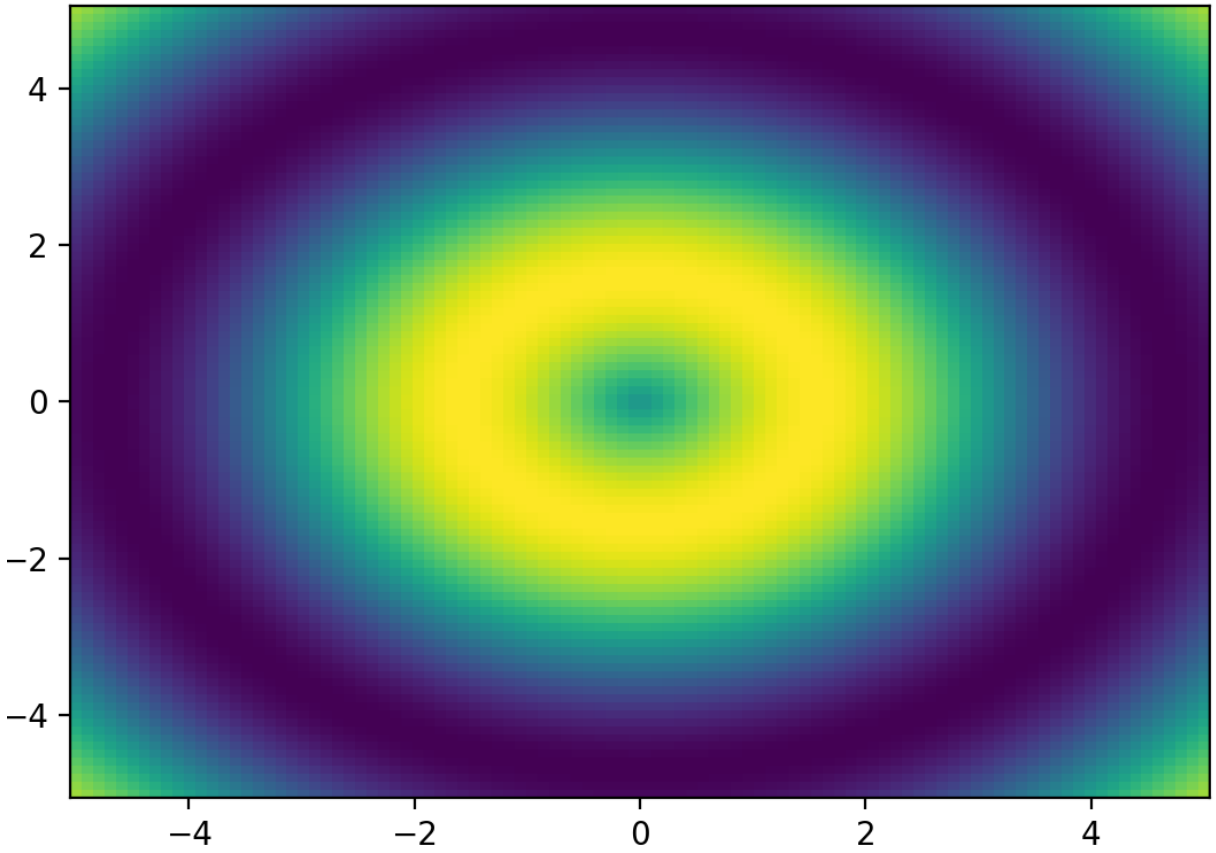
Z = np.sin(np.sqrt(X**2 + Y**2))

plt.pcolormesh(X, Y, Z)
plt.show()
```

Introduction au module matplotlib

Visualiser une fonction à deux variables

Utilisation de `pcolormesh()` : permet une visualisation grâce à des couleurs.



Visualiser une fonction à deux variables

Affichage d'une échelle des couleurs - colorbar():

```
import numpy as np
import matplotlib.pyplot as plt

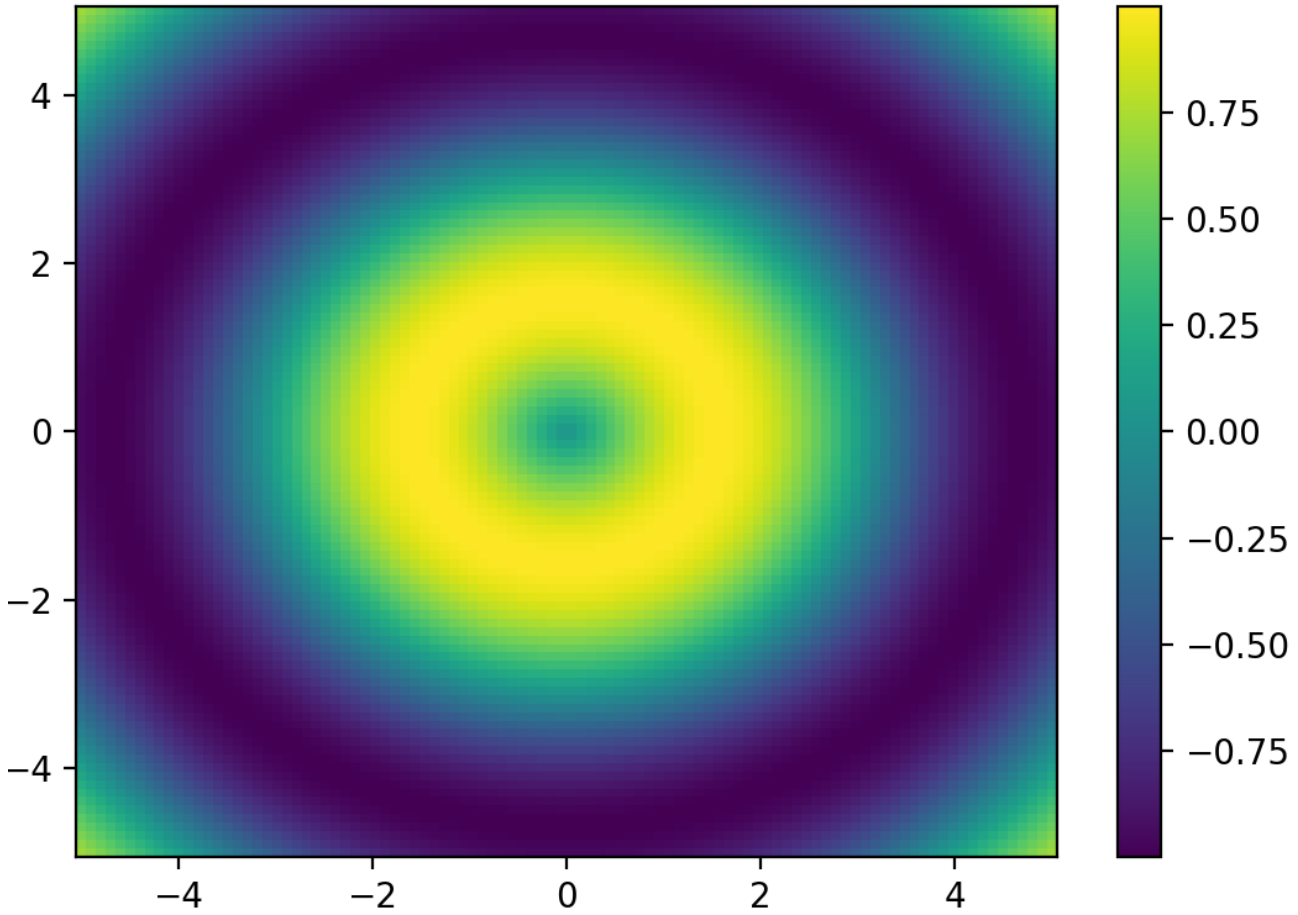
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)

Z = np.sin(np.sqrt(X**2 + Y**2))

plt.pcolormesh(X, Y, Z)
plt.colorbar()
plt.show()
```

Visualiser une fonction à deux variables

Affichage d'une échelle des couleurs - `colorbar()`:



Introduction au module matplotlib

Visualiser une fonction à deux variables

Projection en 3D :

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)

Z = np.sin(np.sqrt(X**2 + Y**2))

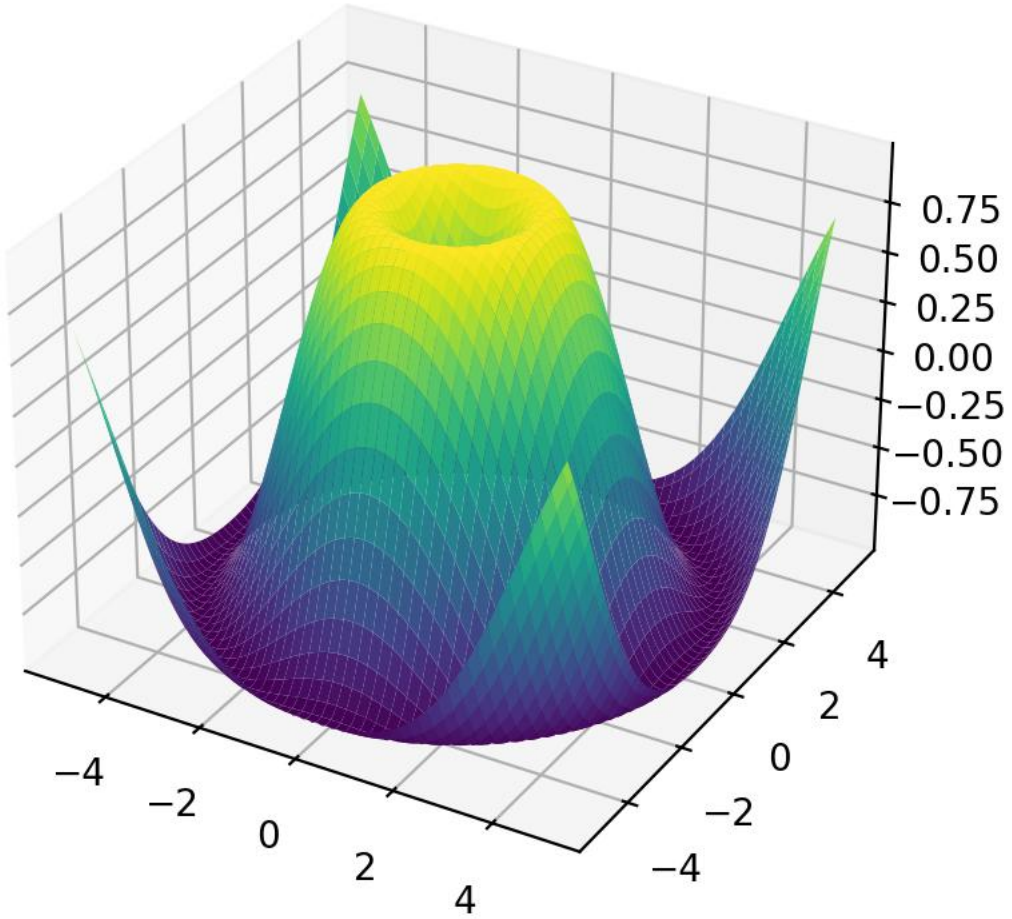
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
ax.plot_surface(x, y, z, cmap='viridis')

plt.show()
```

Introduction au module matplotlib

Visualiser une fonction à deux variables

Projection en 3D :



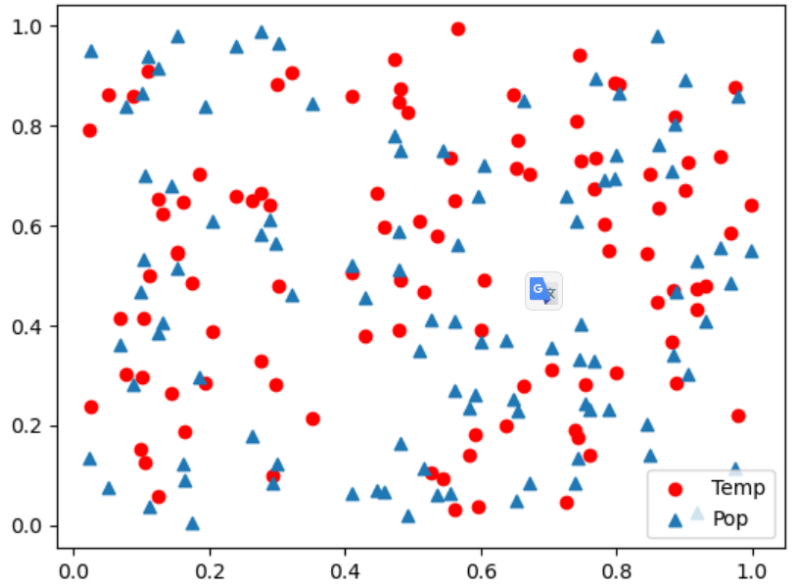
Introduction au module matplotlib

Dessiner des nuages de points

```
N = 100

x = np.random.rand(N)
y1 = np.random.rand(N)
y2 = np.random.rand(N)

plt.scatter(x, y1, marker='o', color='red', label="Temp")
plt.scatter(x, y2, marker='^', label ="Pop")
```



FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

DÉPARTEMENT DE MATHÉMATIQUES ET INFORMATIQUE

Les exceptions

Master : MNSA

Pr: Youssef Ouassit

FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

DÉPARTEMENT DE MATHÉMATIQUES ET INFORMATIQUE

La bibliothèque de modules standards et principaux modules, Modules d'optimisation : SciPy optimizer, NLOpt, CVXPY

Master : MNSA

Pr: Youssef Ouassit

FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

DÉPARTEMENT DE MATHÉMATIQUES ET INFORMATIQUE

Introduction à l'optimisation

Master : MNSA

Pr: Youssef Ouassit