FACULTE DES SCIENCES AIN CHOCK
UNIVERSITE HASSAN II DE CASABLANCA

Département: Mathématiques & Informatique

Chapitre 1: Rappels de base

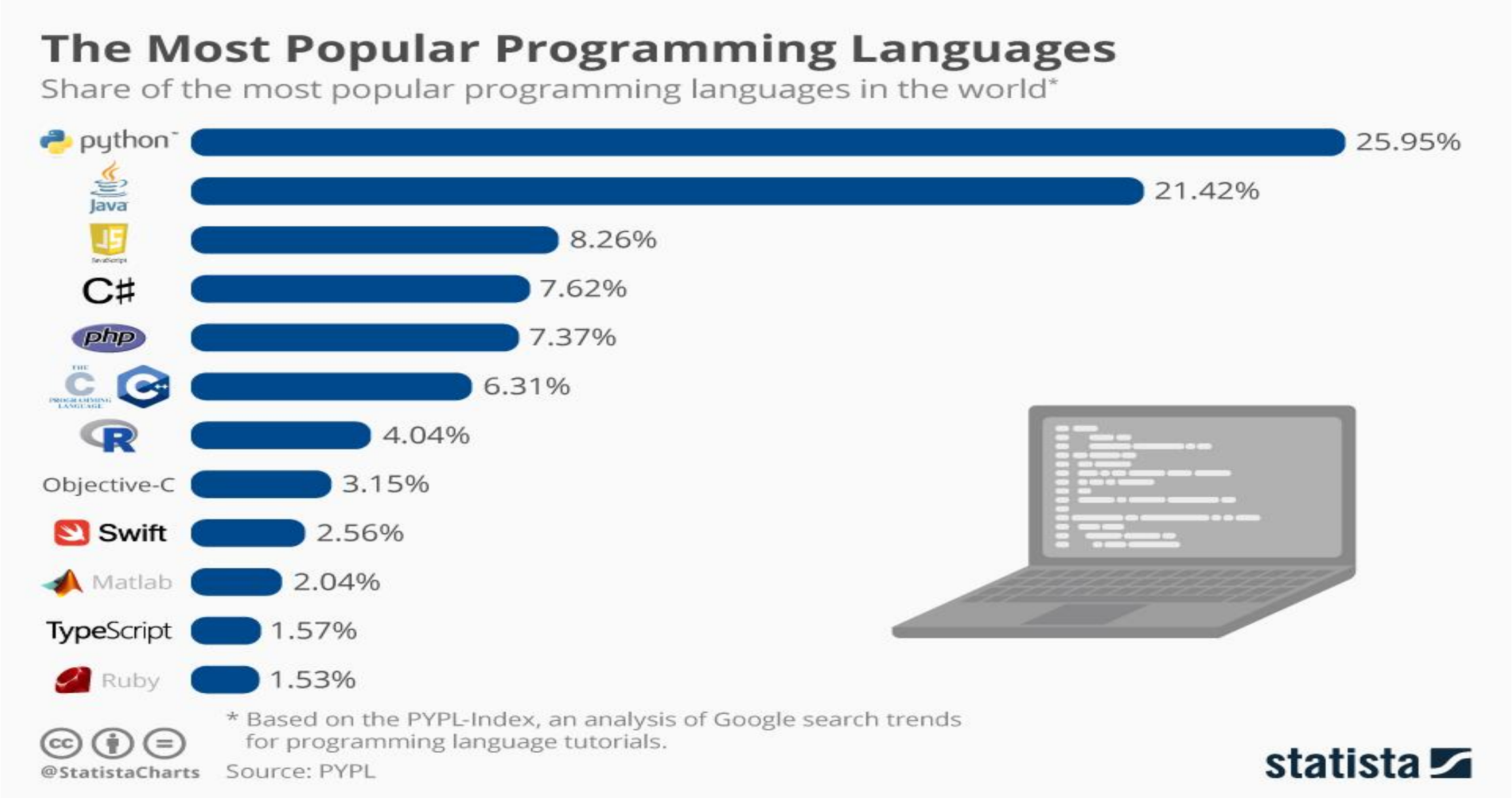
Licence Fondamentale : Tronc commun Informatique Appliquée

Pr: Y.Ouassit

Chapitre 1: Rappels de base

1. Types de données fondamentaux
2. Instructions de bases
3. Structures de contrôles
4. Structures de données
5. Fonctions

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, 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.

Plusieurs implémentation de Python

La spécification VS l'implémentation :

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

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.

Evolution à travers des : 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.
- ***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.
- ***PyPy***: Une implémentation de Python avec un compilateur Just-In-Time (JIT), qui peut rendre l'exécution du code Python plus rapide que CPython.

Plusieurs implémentation de Python

La spécification VS l'implémentation :

Choisir l'implémentation:

```
$ python script.py
```

```
$ jpython script.py
```

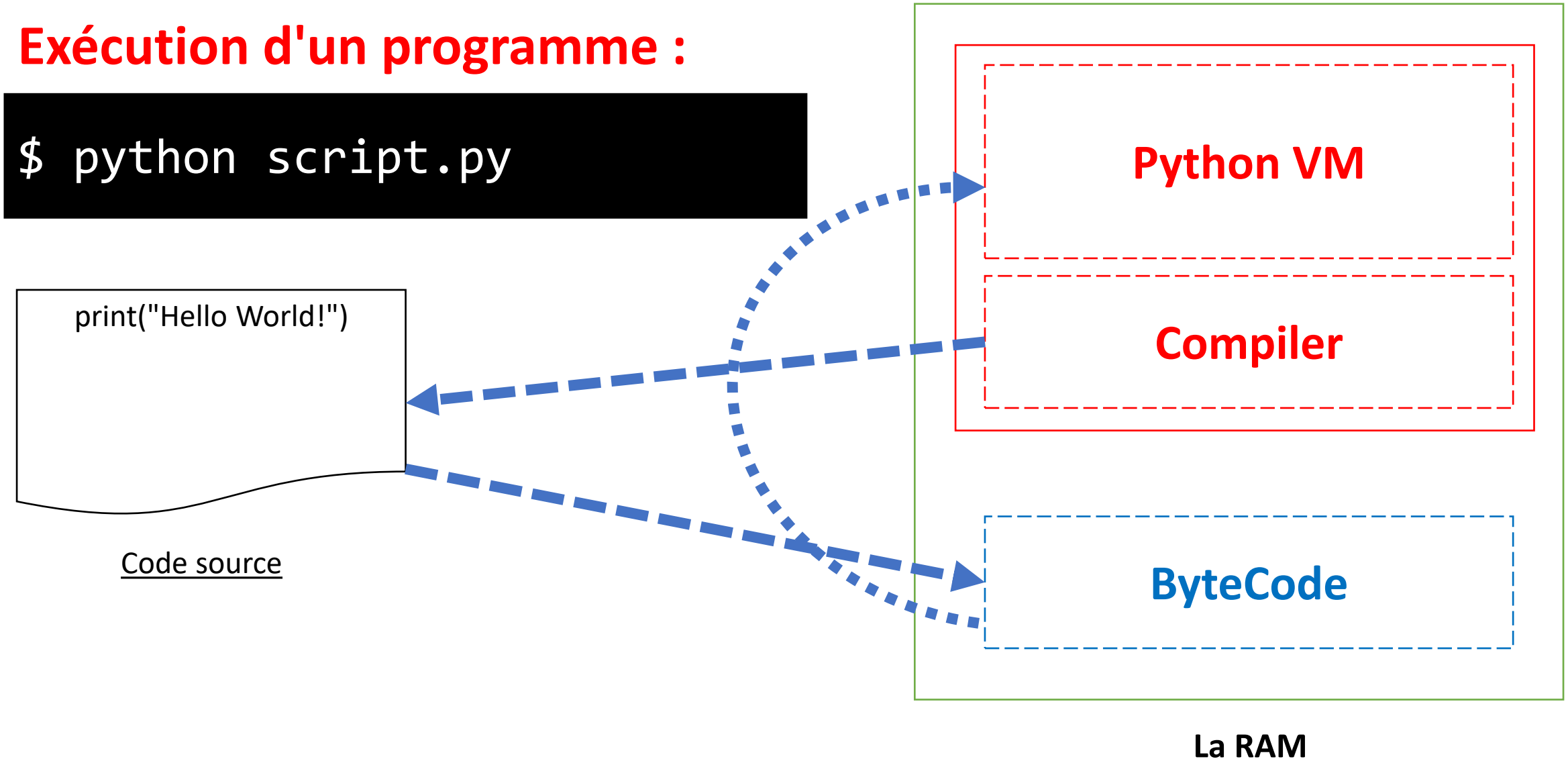
```
$ pypy script.py
```

```
$ ipy script.py
```

Plusieurs implémentation de Python

Exécution d'un programme :

```
$ python script.py
```



Installation de Python

Avant de pouvoir utiliser Python, il faut :

1 - Installer une implémentation du langage, généralement **CPython**, qui est la version officielle et la plus utilisée :

https://www.python.org
https://www.anaconda.com

2 - Installer un IDE :

Syntaxe de Base

Syntaxe de base

Python est un langage de programmation pseudo-compilé. 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

Shells

Python

>>>

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).

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.

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
```

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)

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 chaîne de caractères

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

Enchaînement des instructions

La plus couramment utilisé

1 instruction = 1 ligne

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

Autres possibilités

Peu utilisé

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

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

Opérateurs arithmétiques:

Ces opérateurs sont utilisés pour effectuer des opérations mathématiques.

Opérateur	Description
+	Additionne deux valeurs.
-	Soustrait la seconde valeur de la première.
*	Multiplie deux valeurs.
/	Divise la première valeur par la seconde, renvoie un flottant.
//	Divise la première valeur par la seconde et renvoie un entier (partie entière).
%	Renvoie le reste de la division de la première valeur par la seconde.
**	Élève la première valeur à la puissance de la seconde.

Opérateurs de comparaison:

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

Opérateur	Description
==	Vérifie si les deux valeurs sont égales.
!=	Vérifie si les deux valeurs sont différentes.
>	Vérifie si la première valeur est supérieure à la seconde.
<	Vérifie si la première valeur est inférieure à la seconde.
>=	Vérifie si la première valeur est supérieure ou égale à la seconde
<=	Vérifie si la première valeur est inférieure ou égale à la seconde.

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.

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.

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.

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:")
```

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

Structures conditionnelles

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

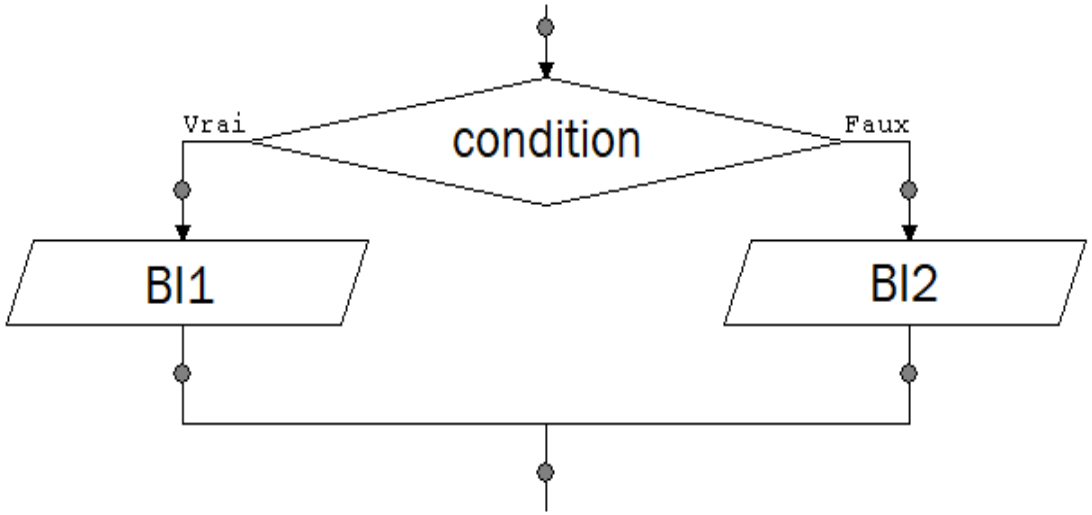
- La structure if
- La structure if … else ….
- La structure if… elif …else
- La structure match

La structure if … else … :

Syntaxe :

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

Organigramme:

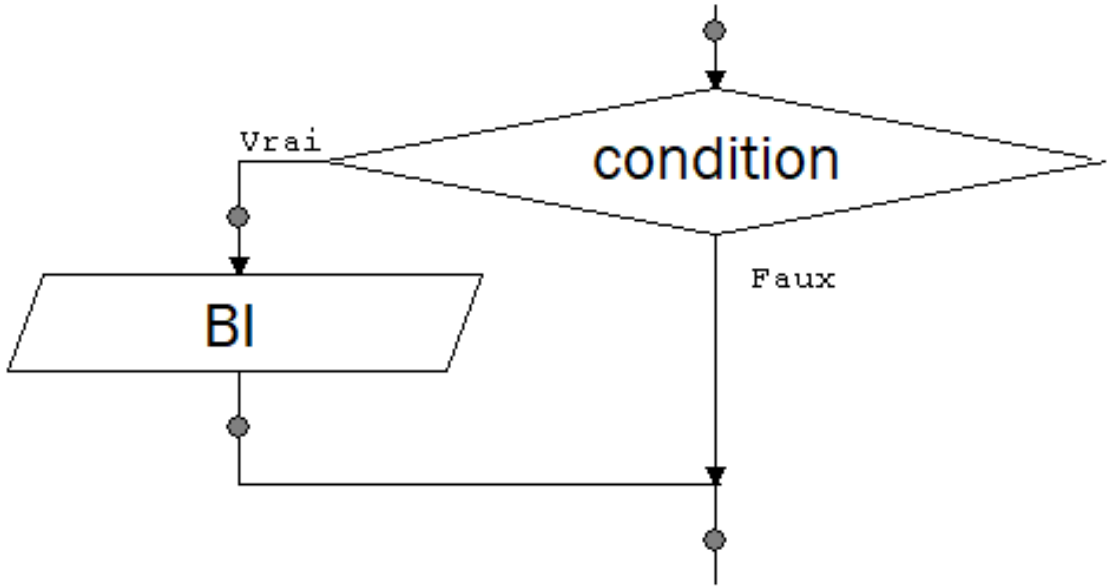


La structure if … :

Syntaxe :

```
if condition:
    BI
```

Organigramme:



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

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:

```
x = 3

match x:
    case 1:
        print("Un")
    case 2:
        print("Deux")
    case 3:
        print("Trois")
    case _:
        print("Autre")
```

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

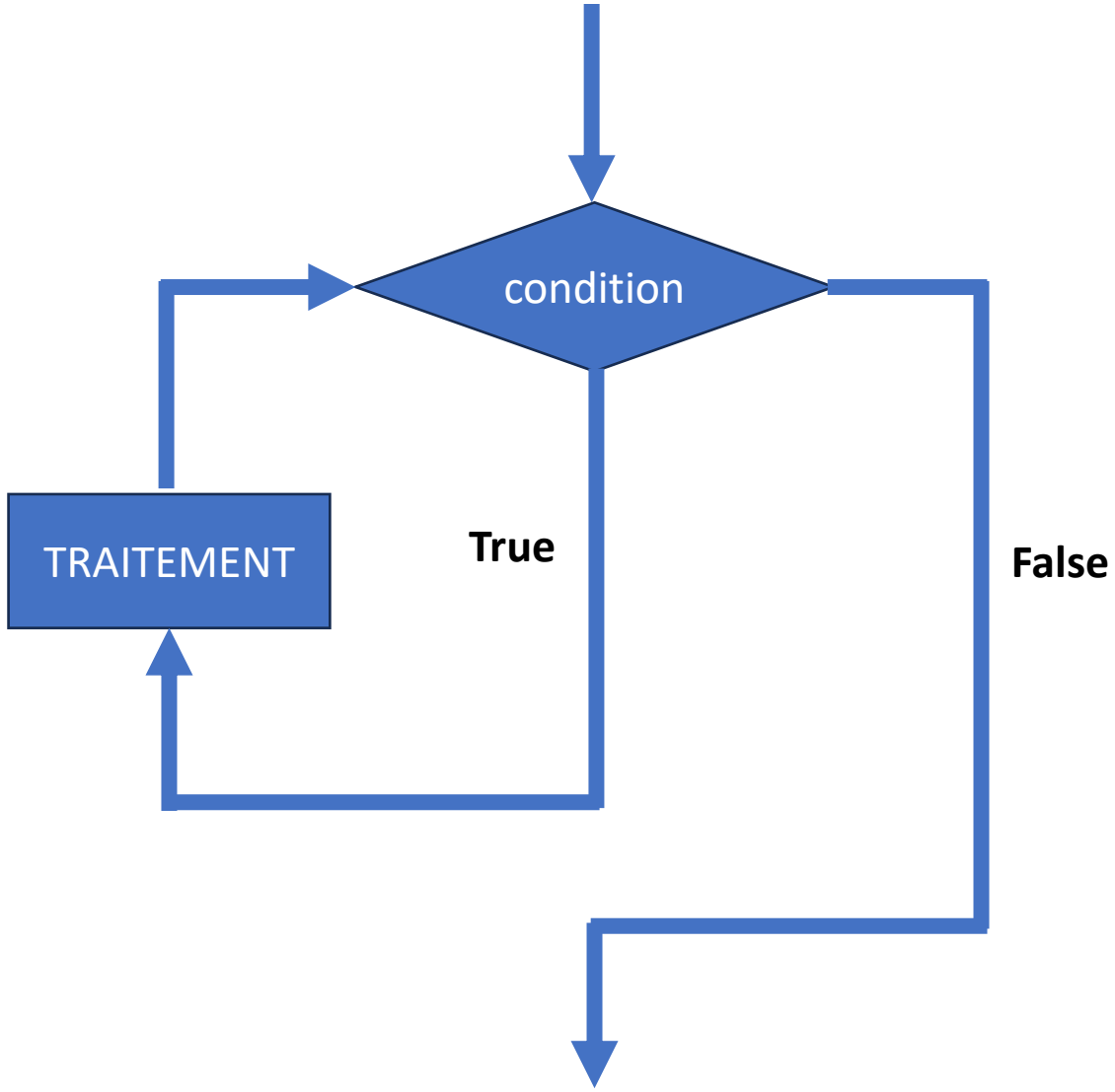
Définition :

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 while avec else

Exemples :

```
i = 1
while i <= 5:
    print(i)
    if i == 3:
        print("Nombre trouvé !")
        break
    i += 1
else:
    print("Aurevoir")
```

```
i = 1
while i <= 5:
    print(i)
    i += 1
else:
    print("Aurevoir")
```

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 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.

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
```

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 donnée

Définition :

Les structures de données essentielles:

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

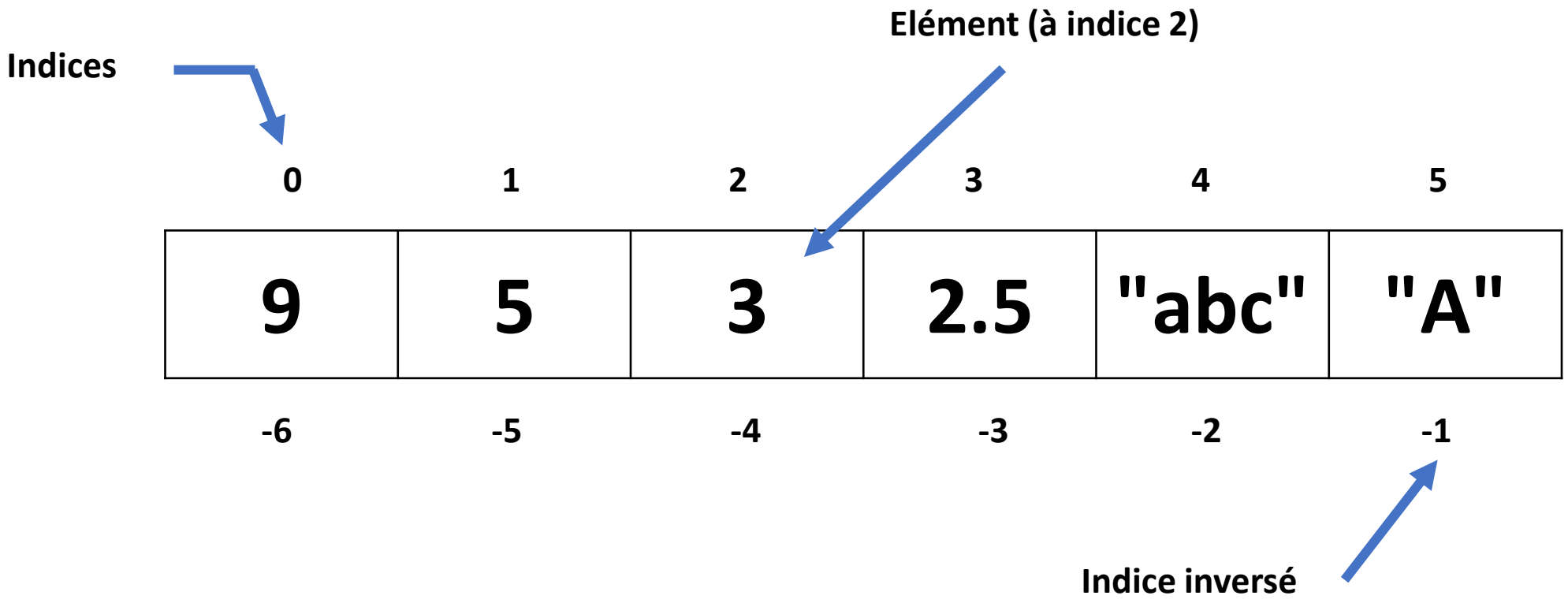
Définition d’une liste

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

```python
# Liste vide
ma_liste_vide = []

ma_liste_vide = list()

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

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

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]
```

Les listes

- Opérateurs
- Fonctions
- Méthodes

Opérateurs de base

Opérateur d'Appartenance **in**

```
fruits = ["pomme", "banane", "cerise"]

print("pomme" in fruits)  # True

print("orange" in fruits)  # False
```

Opérateurs de base

Opérateur de concaténation :

A = [7 , 2 , 4]

7	2	4

B = [9 , 1]

9	1

C = A + B

Résultat :

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

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

Les fonctions

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
```

Les fonctions

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

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 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 méthodes

Méthode `clear()`

Description : Supprime tous les éléments de la liste.

Syntaxe : `liste.clear()`

Exemple :

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

Les méthodes

Méthode `index()`

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

Syntaxe : `liste.index(élément)`

Exemple :

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

Les méthodes

Méthode `count()`

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

Syntaxe : `liste.count(élément)`

Exemple :

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

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 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 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 méthodes

Remarque :

Attention : = ne copie pas une liste

A = [1, 2, 3]

B = A

👉 On ne crée pas une nouvelle liste. On crée simplement une nouvelle référence vers la même liste en mémoire.

Shells

Python

```
>>> A
```

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

```python
# Tuple vide
tuple_vide = ()
tuple_vide = tuple()

# 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

- Opérateurs
- Fonctions
- Méthodes

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)
```

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'
```

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 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 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 fonctions

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

- len
- min
- max
- sum
- sorted
- …

Les str

Définition d’une chaîne de caractères

Le type **str** en Python est une séquence d'unités 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
- Fonctions
- Méthodes

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"
```

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'
```

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 fonctions

On retrouve les même fonctions déjà vu :

- **len(ch)** retourne le nombre de caractères de ch
- max(ch) retourne le caractère de code ascii maximal
- min(ch) retourne le caractère de code ascii minimal

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.

Les fonctions

La fonction eval() permet d’évaluer une chaîne de caractères comme une expression Python :

```
r = eval("2 + 3")

expression = "10 * (5 + 2)"
resultat = eval(expression)

x = 4
eval("x + 6")
```

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

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 méthodes

Méthode : `count`

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

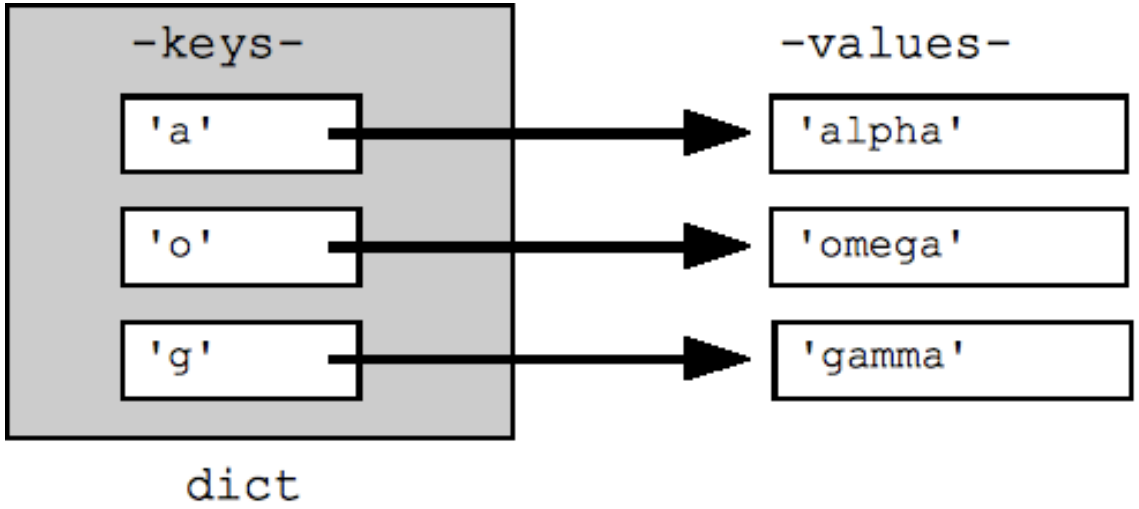
Syntaxe : `str.count(sub[, start[, end]])`

Exemple :

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

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))
```

Les dictionnaires

- Opérateurs
- Fonctions
- Méthodes

Opérateurs de base

Opérateur d'Appartenance **in**

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

Opérateur d’indexation []

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

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"}
```

Les fonctions

<u>On retrouve les même fonctions déjà vu :</u>

- **len(D)** retourne le nombre de d'éléments
- **max(D**) retourne le max des clés
- **min(D)** retourne le min des clés

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

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 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
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 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
```

Plan

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

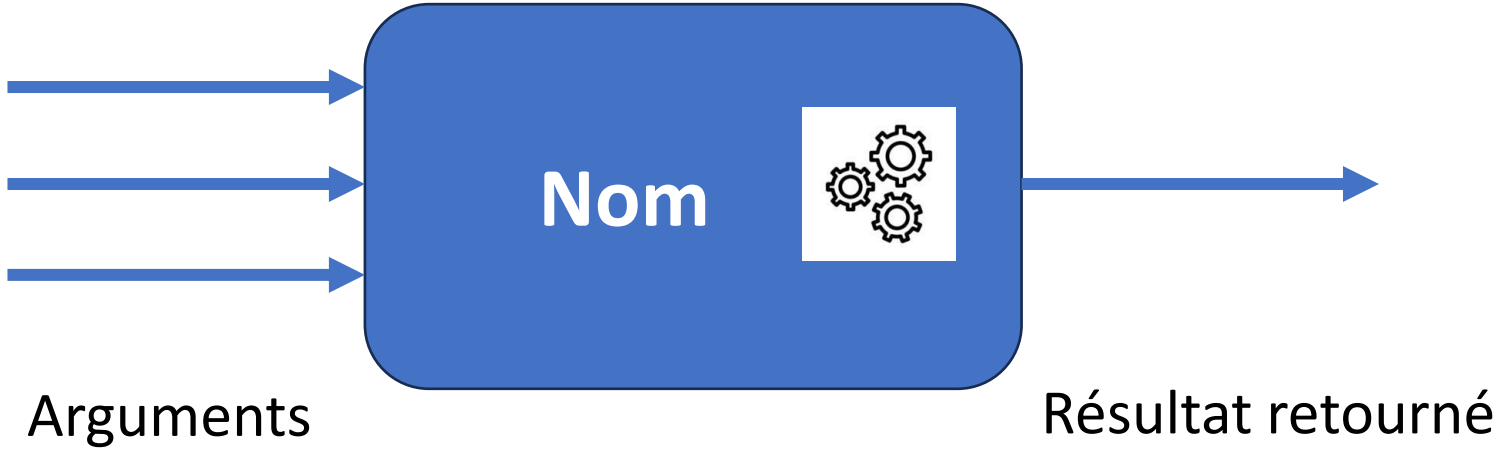
```
n = int(input("Donner n : "))
p = int(input("Donner p : "))
# calculer n!
fn = 1
for i in range(2, n+1):
    fn = fn * i
# calculer p!
fp = 1
for i in range(2, p+1):
    fp = fp * i
```

```
# 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)
```

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.

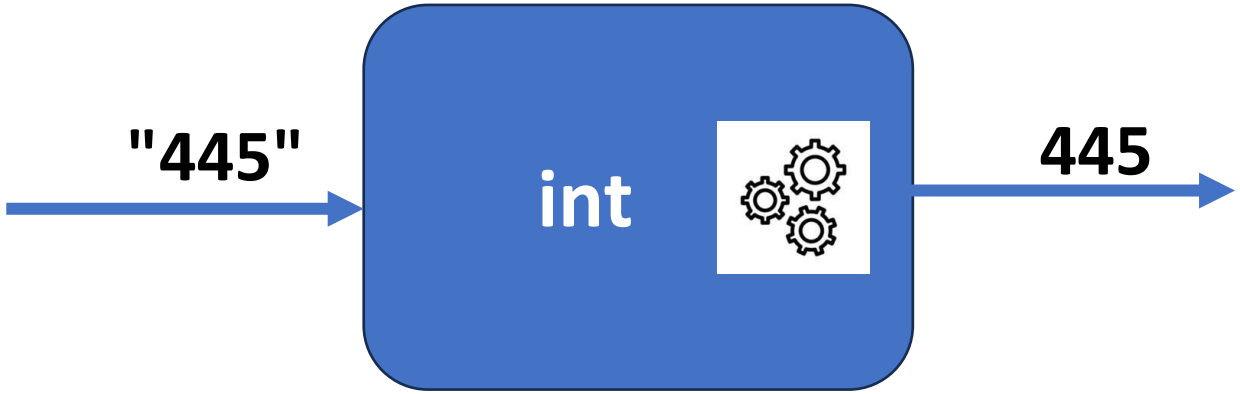


Définition

Fonctions intégrées : sqrt

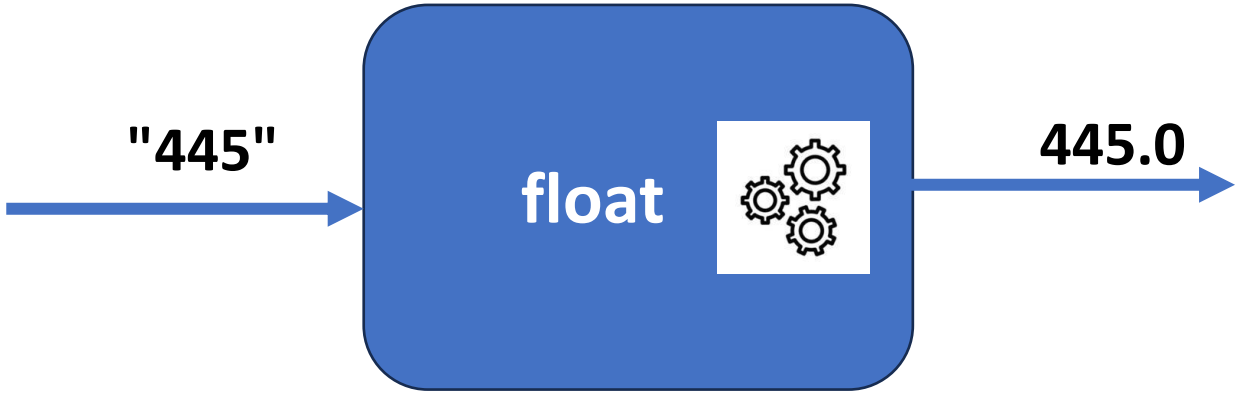
Définition

Fonctions intégrées : int



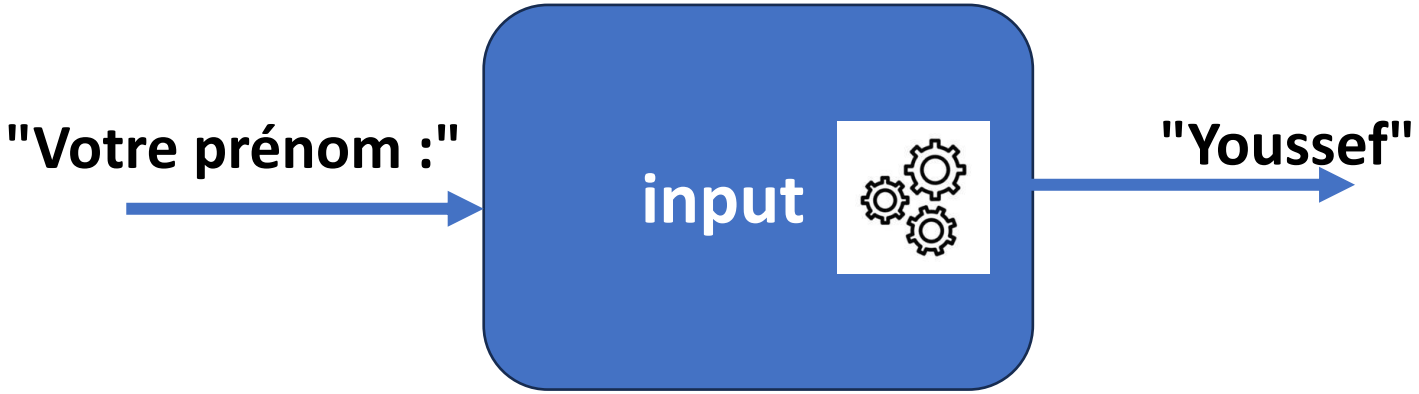
Définition

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
```

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
```

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.

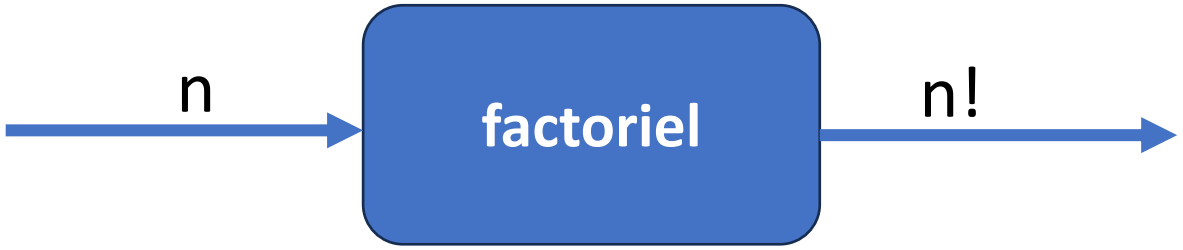
Syntaxe : **variable = nomFonction(Valeur1, Valeur2, …)**

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
```

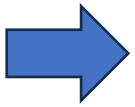
```
M1 = montantTTC(5000 , 10,  0.2)
M2 = montantTTC(7000 , 5  ,  0.2)
M3 = montantTTC(3000 , 10,  0.2)
M4 = montantTTC(100 , 10  ,  0.07)
```

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
```

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.