Chapitre 0 : Rappel

Notions de base en Python

Anis SAIED

IPEIN

2024/25

Plan

Python: Rappel général

- **Documentation officielle**: https://docs.python.org/3/
- EDI: **IDLE**(Python Integrated Development and Learning Environment), Spyder, Pyzo,...
- IDLE > run > shell $\rightarrow$ Mode intéractif
- Le **shell** : programme, execute des lignes de commandes.
- IDLE > new > window/file $\rightarrow$ Mode script (programme .py)
- Chaque programme a 3 flux de données (fichiers standards) associés:
  - stdin (standard input): clavier
  - stdout (standard output): écran
  - stderr (standard error): écran

Plan

Variables

Identificateurs

Le choix de nom de variable ne devra pas être l'un de ces mots clés.

Python keywords

```
for, in, range, while, continue, break, yield
if, elif, else,
or, and, not,
open, close, print, input,
def, global, return, pass, global
lambda, filter, map,
try, except, finally, raise, assert
...
```

Variables

Types

En Python, les variables:

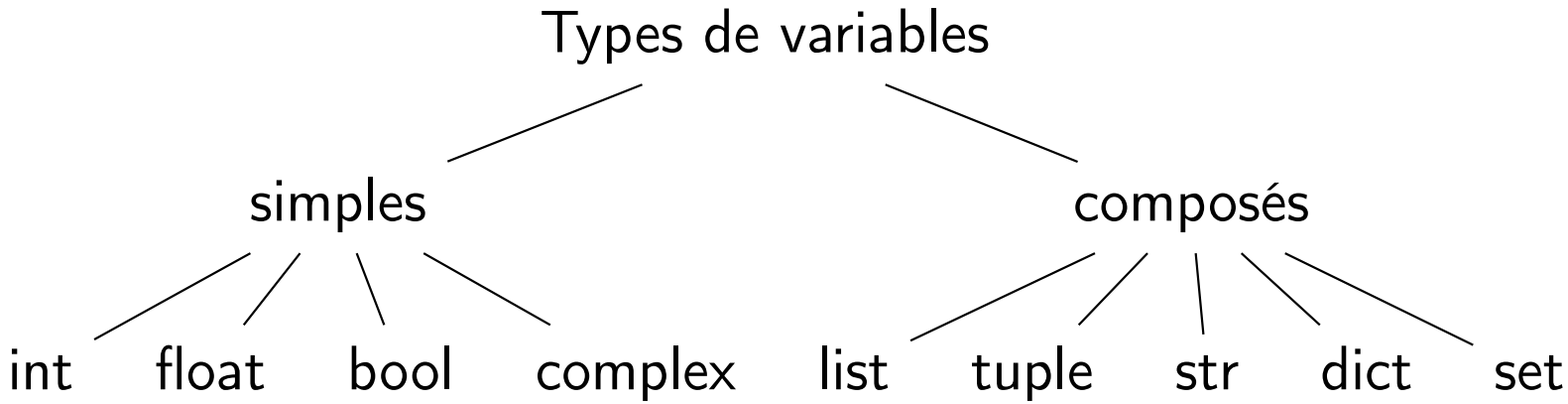
- Non typées
- Le type change selon le contenu
- type(var): retourne le type de la variable

Exemple

```
a = 1
print(type(a)) #int
a = 1.
print(type(a)) #float
```

Variables

Types



Types simples

int & float

- Opérateurs pour `int` et `float`: $+, -, *, /, //$

  `n=1 ; n=n+1 ; n *=2 ; type(n)` → `int`

  `type(n)==int` → `True`

- Notation scientifique

  `f=1e-2` → `0.01`

  `f=3. ; type(f)==float` → `True`

Types simples

bool

- Opérateurs pour le type `bool`: `and, or, not`
- Valeurs possibles : `True, False`

  `a = True ; b = False; c = True + 0 ; d = not a`
- bool(expr) evalue logiquement une expression.
- Toute variable non nulle/vide est évaluée comme `True`.

  `bool("abc") ; bool([1]) ; bool(10)` $\rightarrow$ `True`

  `bool("") ; bool([]) ; bool(0)` $\rightarrow$ `False`

Conversion de types

- dir(type) retourne les fonctions disponibles et applicables sur les valeurs de ce type.

  Exemple : `print(type(int))`

- Convertir une valeur de type **A** en type **B** permet de profiter des fonctions offertes par le type **B**

  `a = 1.0 ; a = int(a)` → passer de `float` à `int`

  `a = int("abc")` → `Erreur de conversion de type`

- `Rappel : Par défaut, toute erreur arrête l'exécution du programme → Il faut prévoyer un traitement alternatif en cas d'erreur`

Gestion d'erreurs

- Toute instruction qui ne respecte pas les règles posées par le langage Python est considérée comme une **Exception**
- Une Exception entraine l'arrêt immédiat du programme
- Eviter l'arrêt imprévisible du programme necessite
  1. Ecrire les instrcutions succeptibles de produire des erreurs (et donc d'arrêter le programme) dans le bloc `try`
  2. Ecrire le traitement alternatif (qui corrige cette éventuelle erreur) dans le bloc `except`
- Poursuivre la suite du programme après la sortie du bloc `try ... except ...`

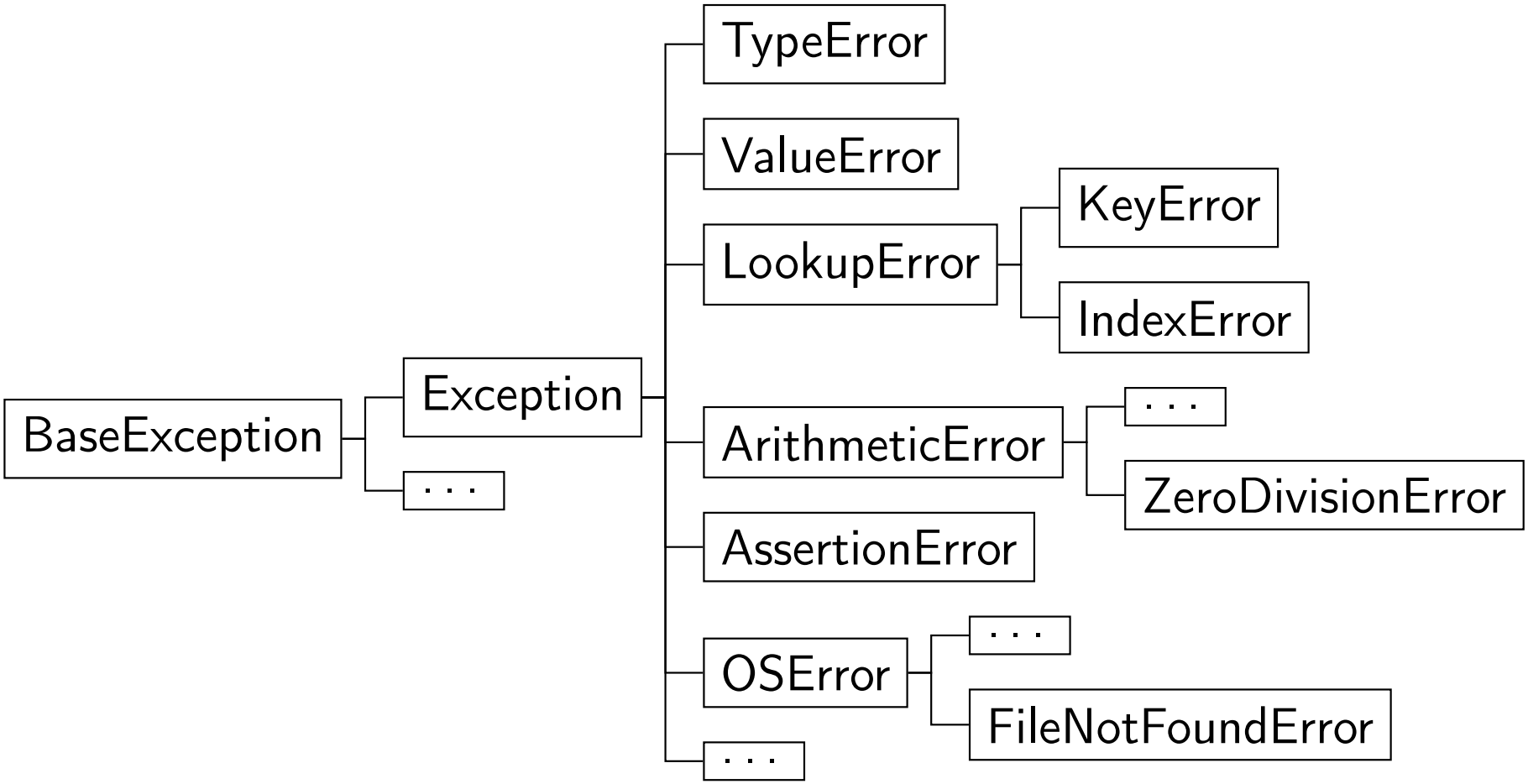
Exemple : voir suite

Gestion d'erreurs

Exemple: try . . . except

```
try:
    x = "12n"
    a = int(x)
    b = a / 0
except ValueError:
    print("Erreur: type attendu 'int'")
except ZeroDivisionError:
    print("Erreur: Division par 0")
except: # toutes les autres erreurs
    print("Erreur imprévue")
else: # si aucune erreur
    print("Pas d'erreur")
finally: # toujours
    print("fin de programme")
```

Principales Exceptions en Python



Liste complete :
https://docs.python.org/3/library/exceptions.html

raise Exception

- On peut décider qu'on a une exception si une expression ne respecte pas la logique et les exigences du problème à résoudre.
- raise permet d'informer Python qu'on est en situation d'exception (lève une exception).

Syntaxe: `raise <type d'exception>`

Exemple

```
try:
    age = -1
    if age < 0:
        raise ValueError
except ValueError:
    print("Erreur: Valeur négative.")
else:
    print("Age correcte.")
```

assert

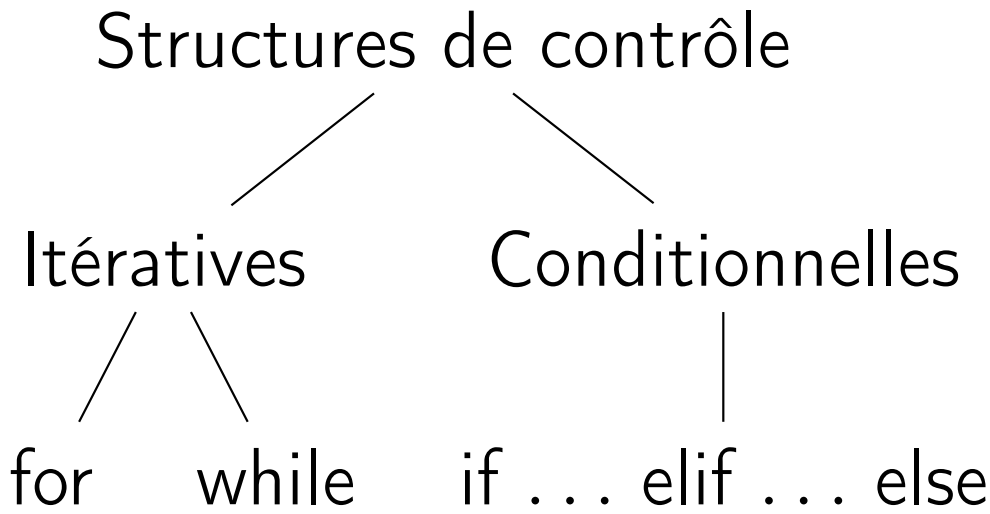
- assert teste si une condition est vraie.
- Si la condition est fausse, elle lève une exception de type `AssertionError`.
- Cette méthode est utile pour vérifier les hypothèses pendant le développement.

Exemple

```
def division(a, b):
    assert b != 0, "Le diviseur ne doit pas être zéro"
    return a / b

try:
    result = division(10, 0)
except AssertionError as error:
    print(f"Erreur: {error}")
else:
    print(f"Résultat: {result}")
```

Structures de contrôle



Les structures de contrôle conditionnelles

La structure conditionelle **if**

Exemple

```
x = 10
if x > 0:
    print("x est positif")
elif x == 0:
    print("x est zéro")
else: # si tout est faux.
    print("x est négatif")
```

Opérateur ternaire

```
r = True if x>0 else False
r = "+" if x>0 else "-" if x<0 else "0"
L = [i if i%2==1 else 0 for i in range(6)]
# L = [0, 1, 0, 3, 0, 5]
```

Les structures de contrôle itératives

La structure itérative **for**

Syntaxe: `for i in range(start, end, step)`

- Utilisée lorsque le nombre d'itérations est connu à l'avance.
- Idéale pour parcourir des séquences comme des listes, des tuples, et des chaînes de caractères.

Exemples

Calcul de la somme des nombres impairs entre 1 à 10.

```
s = 0;
for i in range(1, 10, 2): # pas = 2
    s += i

s = 0
for i in range(1,10): # pas = 1
    if i % 2 == 0:
        continue # passer à l'itération suivante
    s += i
```

range

La fonction **range()** génère une séquence de nombres, et la boucle for itère sur cette séquence.

Syntaxe : `range(start, stop, step)`

- **start**: Valeur initiale (incluse), 0 par défaut.
- **stop**: Valeur finale (exclue), la séquence s'arrête avant ce nombre
- **step**: Incrémentation (optionnel), la différence entre chaque deux nombres dans la séquence. Si omis, le pas par défaut est 1.

Exemple

```
sum(range(1, 10, 2)) # somme des nombres impairs entre 1 à 10
list(range(0,3)) # Convertit range(0, 5) en une liste [0, 1, 2]
r = range(10); print(r[3]) # élément à l'index 3 (affiche 3)
print(r[2:7]) # Affiche la tranche de 2 à 6 : range(2, 7)
min(r); max(r)
if 5 in r:
```

Les structures de contrôle itératives

La structure itérative **while**

Syntaxe: `while condtition`

- Utilisée lorsque le nombre d'itérations dépend d'une condition.
- Pratique pour les situations où l'on doit attendre qu'une condition change.

Exemples

Calcul de la somme des nombres impairs entre 1 à 10.

```
s = i = 0 # affectation multiple
while 2 : # bool(2) ==True
    if i>10: break  # arreter la boucle
    if i%2 : s += i # éq à  i%2 != 0
    i += 1
# autrement
s,i = 0,10;
while i>0:
    s += i if i%2 else 0 # affecter i if bool(i%2)==True
    i -= 1
```

Plan

Opérations Entrée/Sortie

La fonction *print()*

» `help(print)`

```
print(*args, sep=' ', end='\n')
```

Description :

- `*args` : Les objets à afficher, convertis en chaînes de caractères comme le fait `str()`.
- `sep` : Chaîne de séparation entre les objets (par défaut, un espace).
- `end` : Chaîne ajoutée à la fin. Par défaut, un saut de ligne '`\n`'.

Exemples

Impression de plusieurs objets avec un séparateur personnalisé

```
print("Bonjour", "Ipein", "!", sep="---")
```

Utilisation de 'end' pour personnaliser la fin de ligne

```
print("Début de la ligne", end=" ")
print("fin de la même ligne")
```

La fonction *print()*

Exemples

Impression avec conversion de type

```
num = 123
print("Le nombre est " + str(num))
```

Impression avec formatage

```
nom = "Ali"
age = 30
print("Nom : " + nom + ", Âge : " + str(age))
print("Nom : {}, Âge : {}".format(nom, age))
print(f"Nom : {nom}, Âge : {age}")

print("Nom : %s, Âge : %d" % (nom, age))
#%s :chaine, %d: entier, %f:float

print("Valeur de pi : {:.2f}".format(3.14159))
#.2f affiche le nombre avec 2 décimales: 3.14
```

Opérations Entrée/Sortie

La fonction *input()*

Syntaxe : `input(prompt='')`

- Est une fonction intégrée de Python, utilisée pour lire une chaîne de caractères depuis l'entrée standard (le clavier).
- Lorsque la fonction `input()` est appelée, elle affiche le texte du `prompt` (message qui demande d'entrer des informations) à l'utilisateur, sans ajouter de saut de ligne à la fin.
- Retourne les informations saisies comme des chaînes de caractères.

Exemple

Lecture d'une entrée utilisateur

```
nom = input("Entrez votre nom: ")
print(f"Bonjour, {nom}!")
```

La fonction *input()*

Exemples

Lecture et conversion de données numériques

```
age = input("Entrez votre âge: ")
print(type(age)) # str
try:
    age = int(age)  # Conversion de chaîne à entier
except:
    print("Erreur de valeur")
else:
    print(f"Vous avez {age} ans.")
```

Utilisation d'un prompt avec des indications

```
reponse = input("Voulez-vous continuer ? (oui/non): ")
if reponse.lower() == "oui":
    print("Vous avez choisi de continuer.")
else:
    print("Fin du programme.")
```

Plan

Les conteneurs

Les types modifiables: `list`, `dict`, `set`

Type	Indexé	Ordonné	Modifiable
list	✓	✓	✓
dict			✓
set			✓

Exemples

```
L=[i for i in range(10)];  L.append(len(L)); L[0]=L.pop()
for i in range(len(L[:5])):  print(f"L[{i}]={L[i]}")

e = set(L);  e.remove(5)

d = dict();
for i in range(4):
    d.update({chr(65+i+32):chr(65+i)})
# {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D'}
for k,v in d.items():
    print(f"{k}:{v}",end=",") # a:A,b:B,c:C,d:D
```

Les conteneurs

Les types non modifiables: `tuple, str`

Type	Indexé	Ordonné	Modifiable
tuple	✓	✓	
str	✓	✓	

Exemples

```
s="abc" ; s[0]="e" # erreur
id(s) # 140690396193440
s += "d"
id(s) # 140690160486320

t=tuple(range(4)); # (0, 1, 2, 3)
t[0]=4 # 'tuple' object does not support item assignment
t = t[:2],t[-1] # ((0, 1), 3)
```

Plan

Résumé des Types de Fonctions en Python

- **Fonctions intégrées** : Fonctions prédéfinies disponibles par défaut, comme `print()` et `len()`.
- **Fonctions définies par l'utilisateur** : Fonctions créées par les développeurs pour réutiliser du code.
- **Fonctions anonymes (lambda)** : Fonctions sans nom utilisées pour des opérations simples.
- **Fonctions d'ordre supérieur** : Fonctions prenant d'autres fonctions en argument ou renvoyant des fonctions.
- **Fonctions récursives** : Fonctions qui s'appellent elles-mêmes pour résoudre des problèmes divisibles.

Fonctions intégrées

- Fonctions prédéfinies.
- Disponibles sans importation.
- Exemples :
  - `print()` : Affiche du texte à l'écran.
  - `len()` : Retourne la longueur d'une séquence.

Exemple:

```
texte = "Bonjour"
print(len(texte))  # Affiche 7
```

Fonctions définies par l'utilisateur

Une fonction sert à organiser et réutiliser le code et définie à l'aide du mot-clé def.

Exemple:

```
def addition(a, b):
    return a + b

print(addition(3, 5))  # Affiche 8
```

C'est quoi le mot-clé `return` et quand l'utiliser ?

- `return` termine l'exécution d'une fonction et renvoie les données traitées à l'appelant.
- Arrête le flux d'exécution d'une fonction, même sans renvoyer aucune valeur.
- Une même fonction peut avoir zéro ou plusieurs instructions `return`.
- Une fonction sans `return` renvoie `None` par défaut.

Fonctions anonymes

lambda

- Utilisées pour des opérations simples.
- Définies avec le mot-clé `lambda`.

Exemple 1

```
addition = lambda x, y: x + y
print(addition(3, 5))  # Affiche 8
```

Exemple 2

```
def appliquer_fonction(f, liste):
    return [f(x) for x in liste]

resultats = appliquer_fonction(lambda x: 2 * x, [1, 2, 3])
print(resultats)  # Affiche [2, 4, 6]
```

Fonctions d'ordre supérieur

map, filter

- map() Applique une fonction à chaque élément d'une collection et retourne un nouvel itérable contenant les résultats.
- filter() Sélectionne les éléments d'une collection qui satisfont une condition donnée et retourne un nouvel itérable contenant ces éléments.

Exemples

```
# Exemple d'utilisation de map()
nombres = [1, 2, 3, 4, 5]
# Multiplie chaque nombre par 2
double = list(map(lambda x: x * 2, nombres))
print(double)  # Sortie: [2, 4, 6, 8, 10]

# Exemple d'utilisation de filter()
# Sélectionne les nombres pairs
pairs = list(filter(lambda x: x % 2 == 0, nombres))
print(pairs)   # Sortie: [2, 4]
```

Fonctions récursives

- S'appellent elles-mêmes pour résoudre des sous-problèmes.
- Utile pour des problèmes comme le calcul de factorielle.
- Les paramètres de l'appel récurssif convergent vers le cas de base.

Exemple

```
def factorielle(n):
    if n == 0: # cas de base
        return 1
    else:
        return n * factorielle(n - 1) # appel récurssif

print(factorielle(5))  # Affiche 120
```

Paramètres, Arguments

Paramètres & Arguments

Paramètres:

- Ce sont les variables définies dans la **signature** d'une fonction.
- Ils représentent les valeurs que la fonction attend de recevoir lors de son appel.

Exemple:

```
def saluer(nom, age):
    print(f"Bonjour {nom}, vous avez {age} ans!")
```

Arguments:

- Ce sont les valeurs réelles passées à une fonction lors de son appel.

Exemple:

```
saluer("Ali", 30)  # "Ali" et 30 sont des arguments
```

Paramètres, Arguments

Valeurs par Défaut

Valeurs par défaut:

- Permettent de spécifier une valeur par défaut pour un paramètre si aucun argument n'est fourni lors de l'appel de la fonction.
- Utile pour simplifier les appels de fonction lorsque des valeurs courantes sont souvent utilisées.

Exemple:

```
def saluer(nom, age=20):
    print(f"Bonjour {nom}, vous avez {age} ans!")

saluer("Aymen") # age utilise la valeur par défaut de 20
saluer("Sami", 40) # age est explicitement défini à 40
```

Les fonctions: Passage de paramètres

Passage de paramètres : par valeur

Passage par valeur:

- Les valeurs des variables sont copiées dans les paramètres de la fonction.
- Les modifications apportées aux paramètres dans la fonction n'affectent pas les variables d'origine.
- Les conteneurs non modifiables (tuple, str) et les types simples (int, float, ...) sont passés par valeur.

Exemple:

```
def increment(value):
    value += 1
    return value # à qui ?

x = 10
increment(x) # est différent de x = increment(x)
print(x)  # x est toujours 10
```

Les fonctions: Passage de paramètres

Passage de paramètres : par adresse

Passage par adresse

- La fonction reçoit des références aux objets, pas les objets eux-mêmes.
- Les modifications apportées à l'objet dans la fonction affectent l'objet d'origine.
- Tous les conteneurs modifiables sont passés par référence.

Exemple:

```
def append_element(lst, elem):
    lst.append(elem)

my_list = [1, 2, 3]
append_element(my_list, 4)
print(my_list)  # my_list est maintenant [1, 2, 3, 4]
```

Variables locales et globales

Variables locales

Variables locales

- Déclarées à l'intérieur d'une fonction.
- Valables uniquement dans la portée de cette fonction.

Exemple:

```
def f():
    x = 5  # Variable locale
    print(x)

f()
print(x) #Erreur: x n'est pas définie ici
```

Variables locales et globales

Variables globales

Variables globales

- Déclarées à l'extérieur de toute fonction.
- Valables dans tout le script, y compris dans les fonctions.

Exemple:

```
x = 10  # Variable globale
def f():
    print(x) # Peut accéder à la variable globale

f()
print(x) # Accessible ici aussi
```

Variables locales et globales

Variables globales : global

Utilisation de `global`

- Permet de modifier une variable globale à l'intérieur d'une fonction.

Exemple:

```
x = 10
def f():
    global x
    x = 20

f()
print(x)  # 20
```

Sans **global**, on aura deux variables différentes (locale et globale) de mêmes noms mais avec des valeurs différentes.

Identificateurs (ID)

Fonction `id()` en Python:

- Retourne l'identifiant (adresse mémoire) d'un objet.
- Peut être utilisée pour vérifier si deux variables font référence au même objet en mémoire.

Exemple :

```
def check_id(a, b):
    print(f"Identifiant de a: {id(a)}")
    print(f"Identifiant de b: {id(b)}")
    if id(a) == id(b):
        print("a et b font référence au même objet.")
    else:
        print("a et b ne font pas référence au même objet.")

# Exemple d'utilisation
x = [1, 2, 3];    y = x ;    z = [1, 2, 3]
check_id(x, y) # a et b sont identiques, mêmes identifiants.
check_id(x, z) # a et b sont différents objets en mémoire,
# donc leurs identifiants seront différents.
```

Plan

Modules

- **Module** : Un fichier (.py) contenant du code Python.
- **Package** : Un répertoire contenant un fichier `__init__.py` et plusieurs modules.
- **Bibliothèque** : Un ensemble de modules ou de packages.
- La **bibliothèque standard (stdlib)** est un ensemble de modules Python préinstallés avec Python.
- Pour installer un package/module/bibliothèque depuis **PyPI**(Python Package Index), on utilise `pip` (ex: `pip install numpy`).

Avant d'utiliser un module, on l'importe avec la commande `import`.

```
import os
print(os.getcwd())  # Affiche le répertoire de travail actuel

import math as m
print(m.pi)

from random import *
print(randint(0,10))
```

Questions ?