SPUS201 - UE SCIENCES : Introduction à la programmation 2

Denis Dubruel - Cours Magistral N°2

UNIVERSITÉ CÔTE D'AZUR

Année 2024/2025
courriel : prenom.nom@univ-cotedazur.fr

Table des Matières.

Table des matières

Boucles Itératives

Vu dans l'UE précédente :

SCRIPT

```
i : int =0              # initialisation du compteur
while i<5 :             # instruction et condition d'éxécution
    print(2*i,end=" ")
    i=i+1           # incrémentation pour faire évoluer la condition
```

L'interpréteur affiche le double de i pour i variant de 1 à 4 :

SHELL

```
0 2 4 8
>>>
```

Boucles Itératives

Vu dans l'UE précédente :

SCRIPT

```
i : int =0              # initialisation du compteur
while i<5 :             # instruction et condition d'éxécution
    print(2*i,end=" ")
```

L'interpréteur affiche indéfiniment $2 \times i$ pour $i = 0$. La condition $i < 5$ ne change jamais et reste vraie ad-vitam eternam !!!

Nouvelle instruction : **for**

Boucles Itératives

- "**for indice in range**" nouvelle instruction pour itérer

code.py

```
for i in range(3) :   # plus de condition
    print( 2*i )  #plus besoin de faire varier l'indice !
```

SHELL

```
0
2
4
>>>
```

range(3) ou range(0,3) itère sur les entiers 0,1 ,2 mais pas 3 !!

Boucles Itératives

SHELL

```
>>> for k in range(2,8) :
    print(k, end=" ")

2 3 4 5 6 7    # pas(step)de 1

for k in range(2,12,3) :
    print(k, end=" ")

2 5 8 11    # pas de 3
```

SHELL

```
>>> for k in range(8,2,-1) :
    print(k, end=" ")

8 7 6 5 4 3  # pas de -1

for k in range(11,1,-3) :
    print(k, end=" ")

11 8 5 2   # pas de -3
```

range() génère une séquence d'entiers croissants ou décroissants avec une valeur de début, de fin (exclue) et un pas (positif ou négatif).

range(Départ , Fin , Pas)

Boucles Itératives

Rappel :

SHELL

```
>>>  for k in range(12,20,3):
         print(k, end="->")  # pour afficher sur une seule ligne.

12 -> 15 -> 18 ->
```

Afficher les suites ci-dessous avec des boucles **for**.

- 0.0 -> 0.1 -> 0.2 -> ... -> 0.9 -> 1.0
- 0.0 -> 0.5 -> 1.0 -> 1.5 ... -> 19.5 -> 20.0
- 10 -> 8 -> 6 -> ... -> -8 -> -10

Même question avec des boucles **while**

Boucles Itératives - boucle dans boucle

Double boucle imbriquée.

SCRIPT

```
Lettre = "abcdefgh"
Numero = "12345678"
for i in range(7,-1,-1) :
    for j in range (0 ,8,1) :
        print(Lettre[j] + Numero[i] ,end=" ")
    print(' ')            # passage à la ligne
```

Au démarrage i (indice des numéros) vaut 7 , puis j (indice des lettres) varie de 0 à 7. Donc sur une ligne, on affiche une lettre variable puis un numéro constant.

1ere ligne : a8 b8 c8 d8 e8 f8 g8 h8.

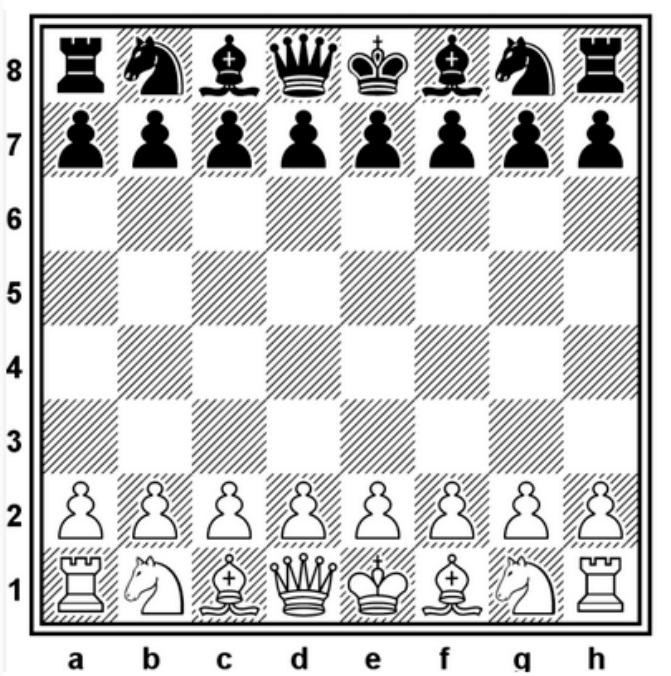
...

8eme ligne : a1 b1 c1 d1 e1 f1 g1 h1 .

Boucles Itératives - boucle dans boucle

SHELL

```
a8 b8 c8 d8 e8 f8 g8 h8
a7 b7 c7 d7 e7 f7 g7 h7
a6 b6 c6 d6 e6 f6 g6 h6
a5 b5 c5 d5 e5 f5 g5 h5
a4 b4 c4 d4 e4 f4 g4 h4
a3 b3 c3 d3 e3 f3 g3 h3
a2 b2 c2 d2 e2 f2 g2 h2
a1 b1 c1 d1 e1 f1 g1 h1
```

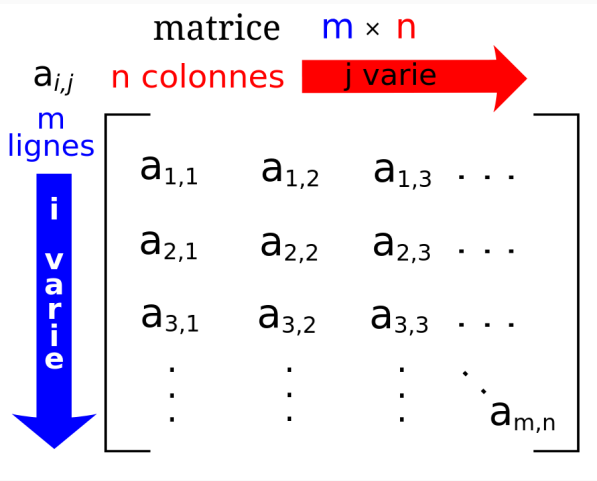


EX : l'ouverture du pion roi , le pion en e2 se déplace en e4.

Boucles Itératives - boucle dans boucle

Exercice : Ecrire le script permettant d'afficher les références des éléments de la matrice A ci-dessous avec la convention rappellée à droite.

$$A = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$



SHELL

```
>>>  affiche( nb_lignes = 3 , nb_colonnes = 4 )
 A(1,1)  A(1,2) ... A(1,4)
 ...
 A(3,1)  A(3,2) ... A(3,4)
```

Boucles Itératives - réductions des boucles

SCRIPT

```
for i in range(1, 11):  # La boucle va de 1 à 10     #L1
    if i % 3 == 0:
        print(f"{i} est divisible par 3.")
        continue # le code va poursuivre la boucle en L1

    if i == 8:
        print(f"{i} est égal à 8, on arrête la boucle.")
        break  # la boucle s'arrête le code va en L11

    print(f"{i} n'est ni divisible par 3 ni égal à 8.")
print('fin de la boucle')                    #L 11
```

Continue : passe à l'itération suivante de la boucle

Break : interrompt ou sort de la boucle

Table des matières

Fonctions - Exemple

SCRIPT

```
def cube(x : float ) -> float : # définition de la fonction
    return x**3                 # retour de la fonction

resultat : float = 0
x: float = 0

for k in range(11) :
    x = k/10
    resultat = cube(x)
    print(f"{x:.2}**3 \u2243  {resultat:.3}") #affichage formatté
                    # \u2243 symbole unicode  pour approximation
```

0.0 * *3 ≈ 0.0
0.1 * *3 ≈ 0.001
...
0.9 * *3 ≈ 0.729
1.0 * *3 ≈ 1.0

Le corps principal du programme appelle ici plusieurs fois la fonction cube qui renvoie une valeur à chaque appel.

Fonctions - Exemple

Ici la fonction ne renvoie pas de valeur explicite, on parle de procédure.(Une procédure renvoie implicitement None)

SCRIPT

```
def afficheResultat(x : float) -> None :
    print(f"{x:2}**3 \u2243  {resultat:.3}")

for k in range(11) :
    x = k/10
    resultat = cube(x)   # la fonction cube renvoie une valeur
    afficheResultat(x)   #
```

Les fonctions et procédures améliorent la lisibilité d'un code, sa compréhension et sa mise au point (debugging).

Fonctions - avantages

Les fonctions permettent :

- d'éviter de recopier des instructions qui se répètent.
- de voir clairement les entrées et sortie :
- d'être réutilisées dans un autre programme :
- d'améliorer la lisibilité du programme complet.
- de décomposer le problème en sous-problème.

```
from math import cos
# importation de la fonction cosinus depuis le module math
from iiwServerHelper import *
# importation de toutes les fonctions (voir TP IP1 !!!)
def addition( a: int, b: int) -> int:
 # nom explicite on voit les entrées et les sorties.
```

Fonctions - exemple de code avec des fonctions simples

```python
from math import pi
def cube(x :float)-> float :
    """ calcule le cube."""
    return x**3

def VolumeSphere(R :float ) -> float :
    """Volume de la sphère."""
    return   4/3*pi*cube(R)

def demandeRayon( ) -> float :
    """ Demande le rayon, """
    ent : str = input ("Rayon en m :")
    return   float(ent)

R : float = demandeRayon( )
V : float = VolumeSphere(R)
print(f"Pour R= {R} m  V= {V:.2f} m³")
```

1) mot clé **def**

2) le type de chague variable d'entrée et sortie est défini.

3) une ligne d'aide """ """ taper help(cube)

4) indentation des commandes

5) chaque fonction est très simple.

6) le corps du programme principal est court !!

Fonctions - tester

Pour tester la fonction : **assert()**

condition vraie : le programme continue

condition fausse : exception de type AssertionError levée, arrêt.

script

```
def cube(x :float)-> float :
    return x**3

assert( cube(2)==8 ) # à écrire avant de coder
assert( cube(0)==1 ) # lève une erreur. (ici faite exprès !)
```

```
Traceback (most recent call last) :
File "/CM2/ex_fonctions.py", line 5, in <module>
        assert(cube(0)==1)
AssertionError
```

Fonctions - tester

- Réfléchir aux tests avant de coder la fonction :
  - aide à rédiger le contenu de la fonction.
  - permet de penser aux cas limites .
- Ecrire les tests justes après la fonction :
  - Si tout est bon à l'éxécution, rien ne s'affiche.
  - Sinon Assertion error et indication du numéro de ligne.
- Ne fonctionne qu'avec des fonctions renvoyant des valeurs. (ex : une procédure avec print() renvoie None !)

TESTER UNE FONCTION SIMPLE EST FORTEMENT RECOMMANDÉ POUR ÉVITER LES RECHERCHES D'ERREUR BEAUCOUP PLUS COMPLEXE ET DONC CHRONOPHAGE PAR LA SUITE !!

Fonctions - exemple de code

```python
def somme_et_produit(a: int, b: int) -> tuple[int, int]:
    """Calcule  somme et produit de deux entiers."""
    somme :int  = a + b
    produit : int = a * b
    return somme, produit


resultat :  tuple[int, int] =(0,0)
resultat = somme_et_produit(3, 4)
print(f"Somme : {resultat[0]}, Produit : {resultat[1]}")
```

fonction portée des variables.

Résolution des noms en local puis global puis interne en fin.

- Portée globale : au niveau du script en cours
- Porté locale : au niveau de la fonction

Interne

Nom préfini : len , open ...

Globale

Nom défini à la base du module/script

Nom déclaré **global**

Locale

Nom affecté dans

une fonction ou module

fonction portée des variables.

SCRIPT

```
def double( n : int ) -> int :
    print(i)    #  i n'est pas défini dans l'espace local de la fonction.
    return 2*n

double(4)
```

SHELL

```
>>>
Traceback (most recent call last):
  ...
    print(i)
NameError: name 'i' is not defined   #( i n'existe nulle part !!)
```

fonction portée des variables.

SCRIPT

```
i : int = 2     # i est défini dans l'espace global
def doubleV2( n : int ) -> int :
    print(f"Dans l'espace local de la fonction i vaut {i}." )
    return 2*n

appel : float = doubleV2(4)
print(f"L appel de la fonction doubleV2(4) renvoie {appel}" )
```

SHELL

```
>>>
Dans l espace local de la fonction i vaut 2.
# i est définit en global et est visible en local dans la fonction.
L appel de la fonction doublev2(4) renvoie 8
```

fonction portée des variables.

SCRIPT

```
def doubleV3( n : int ) -> int :
    i : int = 4 # i est défini dans l'espace local
    print(f"Dans l'espace local de la fonction i vaut {i}." )
    return 2*n

appel : float = doubleV3(4)
print(f"L appel de la fonction doubleV3(4) renvoie {appel}" )
print(f"Dans l'espace global  i vaut  {i}." )
```

SHELL

```
Dans l espace local de la fonction i vaut 4.
L appel de la fonction doubleV3(4) renvoie 8
Traceback (most recent call last):
  ...
    print(f"Dans l'espace global  i vaut  {i}." )
NameError: name 'i' is not defined  # i n'est pas visible en global.
```

fonction portée des variables.

SCRIPT

```
def doubleV4( n : int ) -> int :
    global i  # i est déclarée globale
    print(f"Dans l'espace local de la fonction i vaut {i}.
         Elle va être modifiée à {10*i}" )
    i=10*i
    return 2*n

i : int = 3  # i est déclaré dans cet espace global.
appel : float = doubleV4(4)
print(f"Dans l'espace global  i après appel de la fonction vaut  {i}." )
```

SHELL

```
Dans l espace local de la fonction i vaut 3. Elle va être modifiée à 30.
Dans l'espace global  i après appel de la fonction vaut  30.
```

fonction portée des variables.

Interne

Nom préfini : len , open ...

Globale

Nom défini à la base du module/script

Nom déclaré **global**

Locale

Nom affecté dans

une fonction ou module

Conclusion : Soyez rigoureux !

Table des matières

Manipuler les variables - les listes

♡**Définition** (simple) : Une **liste** est une séquence d'objets python.

Une liste est indéxable et modifiable.

SHELL

```
>>>
listeEntier  : list[int] =[40,50,60]
listeChaine  : list[str] =["mot","50","choucroute \n"]
listeObjet : list =[ 45 , 12.5 , True ,'chaine' ]
>>>listeObjet[2] # Les listes sont indexables :
True
>>>listeObjet[0]=999   # les listes sont modifiables
>>>listeObjet
[ 999 , 12.5 , True ,'chaine' ]
>>>del listeObjet[0]   # suppression d'un élément
>>>listeObjet
[ 12.5 , True ,'chaine' ]
```

Manipuler les variables - les listes

Les indices commencent à zéro ou peuvent être négatifs (pratique pour partir de la fin)

liste	[“A”,	“B”,	“C”,	“D”,	“E”,	“F”,]
indice positif :	0	1	2	3	4	5
indice négatif :	-6	-5	-4	-3	-2	-1

SHELL

```
>>>len(liste)    # longueur de la liste de l'exemple
6
>>> liste = liste + ["G" ]  # concaténation
liste
[''A'',''B'', ''C'', ``D'', ``E'', ''F'',''G'']

>>> tableau : list = 5*[0]   # duplication
tableau
[0, 0, 0, 0, 0]
```

Manipuler les variables - les listes

- vu en IP1 : une liste est une collection d'étiquette (mutable !)
- Attention donc à la copie d'une liste :

SHELL

```
>>> P : list = list(range(10,25,5))
    S = P   # copie des étiquettes et non des valeurs !!
    print(f"Avant modification de S[0] on a : P ={P} et S={S}")
    S[0]=0
    print(f"Après  modification de S[0] on a : P ={P} et S={S}")
Avant modification de S[0] on a : P =[10, 15, 20] et S=[10, 15, 20]
Après  modification de S[0] on a : P =[0, 15, 20] et S=[0, 15, 20]
>>>
```

Manipuler les variables - les listes

SHELL

```
>>> P : list = list(range(10,25,5))
    S : list = list(P)  # création d'une nouvelle liste
    print(f"Avant modification de S[0] on a : P ={P} et S={S}")
    S[0]=0
    print(f"Après  modification de S[0] on a : P ={P} et S={S}")
Avant modification de S[0] on a : P =[10, 15, 20] et S=[10, 15, 20]
Après  modification de S[0] on a : P =[10, 15, 20] et S=[0, 15, 20]

>>>import copy
   Q=copy.copy(P) # utilisation de la méthode copy
   Q[0]=0
   print(f"Après  modification de Q[0] on a : P ={P} et Q={Q}")
Après  modification de Q[0] on a : P =[10, 15, 20] et Q=[0, 15, 20]
```

- La methode **copy()** permet de copier une liste d'objet.

Manipuler les variables - les listes

- La méthode **copy()** : copie liste ”simple”.
- La méthode **deepcopy()** :copie liste de liste.
- Avec une ou plusieurs boucles : algorithmique valable dans d’autres langages (à faire en TP par exemple).

SHELL

```
>>>import copy   # a tester par vous même
   A  : list [ list[int] ] = [ [12 , 15 ] , [ 11 , 28] ]
   B = copy.deepcopy(A)
   B[0][0] = 999
   print(f"Après  modification de B[0][0] on a : A ={A} et B={B}")
```

Manipuler les variables - les tuples

- ♡**Definition** : Les n-uplet ou tuple sont des séquences d'objets.
- Les tuples sont indexables, immutables (non modifiables), concatenables et duplicables !
- les tuples

SHELL

```
>>> collection =('voiture','moto','vélo')
>>>collection[0]   # indexation possible
'Voiture'
>>> collection= 2*collection # duplication
>>> collection
('voiture','moto','vélo','voiture','moto','vélo')
>>>collection=collection+('avion',)  # Notez la virgule !!
>>>collection     # résultat de la concaténation
('voiture','moto','vélo','voiture','moto','vélo','avion')
```

Manipuler les variables - Synthèse

♡Les séquences de données sont des collections ordonnées avec des propriétés communes :

	Chaines	Listes	Tuples
indexation	OUI	OUI	OUI
concateénation	+	+	+
duplication	*	*	*
mutation	NON	OUI	NON
longueur	len()	len()	len()
recherche	in	in	in

Table des matières

Manipuler les variables - recherche dans séquences.

Script

```
for k in range(len(sequence) ) :
        print(sequence[k])
```

Script

```
for objet  in sequence :      # plus explicite
        print(objet)
```

Les 2 lots de commandes affichent le même résultat avec le mot clé **in** il est possible de s'affranchir des indices.

Manipuler les variables - recherche dans séquences

SHELL

```
sequence : str ='titi'
for k in range(len(sequence)):
    print(sequence[k] ,end=" ")

print()

sequence : list = ['o',2,'z']
for k in range(len(sequence)):
    print(sequence[k],end=" ")

print()

sequence : tuple = (3 ,14,'p')
for k in range(len(sequence)):
    print(sequence[k],end=" ")
```

SHELL

```
sequence : str ='titi'
for objet in sequence:
    print(objet  ,end=" ")

print()

sequence : list = ['o',2,'z']
for objet in sequence:
    print(objet  ,end=" ")

print()

sequence : tuple = (3 ,14,'p')
for objet in sequence:
    print(objet  ,end=" ")
```

A vous d'essayer...et de rendre vos codes plus lisibles.

Manipuler les variables - recherche dans séquences

L'instruction `objet in sequence` est un booléen.

script

```
sequence : str ='titi'

trouve:bool=False

for k in range(len(sequence)):
    if 'c'== sequence[k] :  # recherche du caractère c dans la chaîne.
        trouve = True

trouve = ( 't' in sequence ) # nettement plus explicite et court !
```

Licence

Ce document est publié sous licence Creative Commons :

- ©2024 — Denis Dubruel - Université Côte d'Azur
- Attribution
- Utilisation non commerciale
- Partage dans les mêmes conditions 4.0 International

https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode.fr