SPUS201 - UE SCIENCES : Introduction à la programmation 2

Denis Dubruel - Cours Magistral N°4

Année 2024/2025

courriel : prenom.nom@univ-cotedazur.fr

Table des Matières.

Table des matières

Message d'erreur

ERREUR : problème empêchant l'exécution du programme dès son interprétation.

- Erreur de syntaxe (SyntaxError) :
- IndentationError :indentation non respectée.

SHELL

```
>>> %Run Exemple_Erreur.py
Traceback (most recent call last):
  File "/home/Exemple_Erreur.py", line 2
    for k in range(10)
                   ^
SyntaxError: expected ':'
>>> %Run Exemple_Erreur.py
Traceback (most recent call last):
  File "/home/Exemple_Erreur.py", line 6
    i : int = input('Age ?')
IndentationError: unexpected indent
```

"Facile et rapide à corriger !" => lire le message en entier !

Message d'erreur

Erreur ou pas ???

```
1 if n==2  or m==3
2     print("Hello")
```

Message d'erreur

```
1 if n==2  or m==3
2     print("Hello")
```

SHELL

```
1 >>> %Run Ex_ereur.py
2 Traceback (most recent call last):
3   File "/home/Ex_ereur.py", line 2
4     if n==2  or m==3
5                     ^
6 SyntaxError: expected ':'
```

Message d'erreur

```
1 if n==2  or m==3
2     print("Hello")
```

SHELL

```
1 >>> %Run Ex_ereur.py
2 Traceback (most recent call last):
3   File "/home/Ex_ereur.py", line 2
4     if n==2  or m==3
5                      ^
6 SyntaxError: expected ':'
```

Correction :

```
1 if n==2  or m==3   :
2     print("Hello")
```

Message d'erreur

Erreur ou pas ???

```
1 def ma_super_fonction_bien( n: int, message :str, dico : dict )-> list[str]:
2 print("Hello")
```

Message d'erreur

```
def ma_super_fonction_bien( n: int, message :str, dico : dict )-> list[str]:
print("Hello")
```

Interpréteur python :

SHELL

```
>>> %Run Ex_ereur.py
Traceback (most recent call last):
  File "/home/Ex_ereur.py", line 6
    print("Hello")
    ^^^^^

IndentationError: expected an indented block after function definition
 on line 5
```

Message d'erreur

```
1 def ma_super_fonction_bien( n: int, message :str, dico : dict )-> list[str]:
2 print("Hello")
```

SHELL

```
1 >>> %Run Ex_ereur.py
2 Traceback (most recent call last):
3   File "/home/Ex_ereur.py", line 6
4     print("Hello")
5     ^^^^^
6 IndentationError: expected an indented block after function definition
7  on line 5
```

Correction :

```
1 def ma_super_fonction_bien( n: int, message :str, dico : dict )-> list[str]:
2     print("Hello")
```

Message d'erreur

Erreur ou pas ???

```
1 from module_EX_3_TP1 import demander_entier as entrer_entier
```

Message d'erreur

```
1 from module_EX_3_TP1 import demander_entier as entrer_entier
```

Interpréteur python :

SHELL

```
1 >>> %Run Ex_ereur.py
2 Traceback (most recent call last):
3   File "/home/denis_admin/00_Universite/Cours_A_FAIRE_IP_Semestre_2/Cours_DD_IP2_v2
4     from module_EX_3_TP1 import demander_entier as entrer_entier
5 ModuleNotFoundError: No module named 'module_EX_3_TP1'
```

Message d'erreur

```
from module_EX_3_TP1 import demander_entier as entrer_entier
```

SHELL

```
>>> %Run Ex_ereur.py
Traceback (most recent call last):
  File "/home/Ex_ereur.py", line 1, in <module>
    from module_EX_3_TP1 import demander_entier as entrer_entier
ModuleNotFoundError: No module named 'module_EX_3_TP1'
```

Correction 1/2 : commencer par essayer :

```
#from module_EX_3_TP1 import demander_entier as entrer_entier
import module_EX_3_TP1
```

Afin de voir si le module est bien dans le répertoire avec la bonne synthaxe.

Message d'erreur

```
1 from module_EX_3_TP1 import demander_entier as entrer_entier
```

SHELL

```
1 >>> %Run Ex_ereur.py
2 Traceback (most recent call last):
3   File "/home/Ex_ereur.py", line 1, in <module>
4     from module_EX_3_TP1 import demander_entier as entrer_entier
5 ModuleNotFoundError: No module named 'module_EX_3_TP1'
```

Correction 2/2 : Vérifier la frappe ou en externe au code a minima :

1. Vérifier la synthaxe
2. Chercher...aide en ligne...(pratiquer python fréquemment aide)

Message d'erreur

Erreur ou pas ???

```
1 class = 5
2 print =12
3 len = 15
4 range =1
```

Message d'erreur

```
1 class = 5
2 print =12
3 len = 15
4 range =1
```

Interpréteur python :

SHELL

```
1 >>> %Run Ex_ereur.py
2 Traceback (most recent call last):
3   File "/home/Ex_ereur.py", line 14
4     class = 5
5           ^
6 SyntaxError: invalid syntax
```

Message d'erreur

```
1 class = 5
2 print =12
3 len = 15
4 range =1
```

SHELL

```
1 >>> %Run Ex_ereur.py
2 Traceback (most recent call last):
3   File "/home/Ex_erreur.py", line 14
4     class = 5
5           ^
6 SyntaxError: invalid syntax
```

Correction : Mot clé réservé, changer "class" par autre dénomination.

Message d'erreur

Erreur ou pas ???

```
1 classe_nb = 5
2 print =12
3 len = 15
4 range =1
```

Message d'erreur

```
1 classe_nb = 5
2 print =12
3 len = 15
4 range =1
```

Interpréteur python :

⚠Pas d'erreur mais vous vous privez de certaines fonctions !!!

Il est judicieux de ne pas utiliser des mots de commandes (qui apparaissent colorées dans l'éditeur).

Message d'erreur

Erreur ou pas ???

```
1 x : float = 14//2 + 5**3 -4/3
2 # mise à jour :
3 5 = x
```

Message d'erreur

```
1 x : float = 14//2 + 5**3 -4/3
2 # mise à jour :
3 5 = x
```

Interpréteur python :

SHELL

```
1 >>> %Run Ex_ereur.py
2 Traceback (most recent call last):
3   File "/home/Ex_erreur.py", line 3
4     5 = x
5     ^
6 SyntaxError: cannot assign to literal here. Maybe you meant '==' instead
7 of '='?
```

Message d'erreur

```
x : float = 14//2 + 5**3 -4/3
# mise à jour :
5 = x
```

SHELL

```
>>> %Run Ex_ereur.py
Traceback (most recent call last):
  File "/home/Ex_erreur.py", line 3
    5 = x
    ^
SyntaxError: cannot assign to literal here. Maybe you meant '==' instead
of '='?
```

Correction possible : (le signe = est une affectation !)

```
x : float = 14//2 + 5**3 -4/3
# mise à jour (dépend du contexte):
x = 5    # pour une affectation par exemple
#x ==5    # pour obtenir True ou False, à vous de voir.
```

Message d'erreur

CONCLUSION : Les erreurs plantent le code dès le lancement.
Uniquement la première erreur est signalée.

- L'être humain devant le clavier doit lire EN ENTIER l'exception.
- La ligne du code causant l'anomalie est souvent indiquée.
- Il est parfois possible de cliquer dessus pour aller sur la ligne à corriger.

A pratiquer...

Message d'erreur

EXCEPTION type particulier d'erreur qui se produit pendant l'exécution du programme.

```
1 # code générant une erreur.
2 x : float = 0
3 inverse : float = 1/x
```

- Le développeur doit lire EN ENTIER l'exception. La ligne du code causant l'anomalie est indiquée : ici ligne 3.
- le développeur doit lire le message d'exception ici : division by zero.

SHELL

```
1 >>> %Run Exemple_Erreur.py
2 Traceback (most recent call last):
3   File "/home/CM4/Exemple_Erreur.py", line 3, in <module>
4     inverse : float = 1/x
5 ZeroDivisionError: division by zero
```

Exceptions les plus fréquentes

Exceptions fréquentes :

- ArithmeticError
- ZeroDivisionError
- AssertionError
- AttributeError (à venir avec les objets)
- EOFError (End of File , pour les fichiers)
- FileNotFoundError
- IOError (erreur d'entrée ou sortie)

A traiter en fonction de votre code.

Exceptions les plus fréquentes

Exceptions très et même très très fréquentes liées aux types et valeurs :

- IndexError : Index hors limites dans une liste ou un tuple
- KeyError : Clé introuvable dans un dictionnaire.
- ValueError : Valeur incorrecte pour un argument ou une conversion.
- TypeError : Type d'argument invalide pour une opération ou fonction.
- NameError : Nom de variable non défini.

KeyboardInterrupt : Interruption par l'utilisateur (Ctrl+C).

assert

Il est possible de générer une AssertionError pour indiquer ce qu'attend le programme.

Si pas d'exception alors pas de message assertionError

```
# assert
# On impose L non vide
def moyenne(L):
    assert L != [], "Liste vide!"
    return sum(L)/len(L)
```

SHELL

```
>>> moyenne([])
  Traceback (most recent call last):
    File "/home//Ex_assert.py", line 4,
     in moyenne
      assert L != [], "Liste vide!"
  AssertionError: Liste vide!
```

```
def moyenne(L):
    assert L != [], "Liste vide!"
    return sum(L)/len(L)

assert( moyenne([0,0]) ==0 )   # test de la fonction.
assert( moyenne([8,12]) ==10 )
assert (0.33 <  moyenne([0,0,1])  and moyenne([0,0,1]) <0.34))
```

Interception et gestion des exceptions

Exemple d'exception levée correctement :

```
raise AssertionError('Il faut lire les consignes !!!')
```

```
>>>
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Il faut lire les consignes !!!
```

Interception et gestion des exceptions

```
1     raise ZeroDivisionError("Moyenne d'une liste vide impossible")
```

```
1 >>> %Run Ex_exception.py
2 Traceback (most recent call last):
3   File "/home/Ex_exception.py", line 1
4     raise ZeroDivisionError("Moyenne d'une liste vide impossible")
5 IndentationError: unexpected indent
```

Qui voit l'anomalie dans l'exception ??? Astuce N°1 : Bien lire tout le message !

Interception et gestion des exceptions

```
1 raise ValueError('L argument doit être pair' + 1)
```

```
1 >>> %Run Ex_exception.py
2 Traceback (most recent call last):
3   File "/home/Ex_exception.py", line 1, in <module>
4     raise ValueError('L argument doit être pair' + 1)
5 TypeError: can only concatenate str (not "int") to str
```

Qui voit l'anomalie dans l'exception ??? Astuce N°2 : Bien lire tout le message !

Interception et gestion des exceptions

```
1 raise IndexError("Je croyais la liste plus grande" + srt(1))
```

```
1 >>> %Run Ex_exception.py
2 Traceback (most recent call last):
3   File "/home/Ex_exception.py", line 1, in <module>
4     raise IndexError("Je croyais la liste plus grande" + srt(1))
5 NameError: name 'srt' is not defined
```

Qui voit l'anomalie dans l'exception ??? Astuce N°3 : Bien lire tout le message !

Interception et gestion des exceptions

Il est possible d'intercepter une exception et de la gérer pour éviter l'arrêt brutal du programme en respectant la trame suivante :

```
try:
    #
    #  code pouvant lever une exception
    #
except :
    #
    #  code s'éxécutant UNIQUEMENT en cas d'exception.
    #

#
# suite normale des instructions.
#
```

Interception et gestion des exceptions

Il est possible d'intercepter une exception et de la gérer pour éviter l'arrêt brutal du programme. Exemple :

```
try:
    y = 1  /  0    # ça plante à coup sûr !
    print("Première mondiale : division par zéro réussie !")
except :
    print("Bien tenté, mais la division par zéro"
          " n'est pas possible !")


print("Passons aux choses sérieuses.")
```

Interception et gestion des exceptions

Il est possible d'intercepter une exception et de la gérer pour éviter l'arrêt brutal du programme.

```
try:
    y = 1  /  0    # ça plante à coup sûr !
    print("Première mondiale : division par zéro réussie !")
except :
    print("Bien tenté, mais la division par zéro"
          " n'est pas possible !")


print("Passons aux choses sérieuses.")
```

```
>>> %Run EX_try.py
Bien tenté, mais la division par zéro n est pas possible !
Passons aux choses sérieuses.
```

Interception et gestion des exceptions

```
1  def f():
2  # déclenche une exception mais ne la rattrape pas
3      2/0
4      print('Message dans f')
5
6  def g():
7      f() # déclenche une exception
8      print('Message : Dans g !')
9
10 def h():
11     try:
12         g()
13         print('Message : Après g !')
14     except:
15         print('Message : Exception attrapée !')
16 h()
```

Quel va être le résultat de l'éxécution de ce code ??

Interception et gestion des exceptions

```
1  def f():
2  # déclenche une exception mais ne la rattrape pas
3      2/0
4      print('Message dans f')
5  def g():
6      f() # déclenche une exception
7      print('Message : Dans g !')
8  def h():
9      try:
10         g()
11         print('Message : Après g !')
12     except:
13         print('Message : Exception rattrapée !')
14 h()
```

```
1  >>> %Run EX_try.py
2  Message : Exception rattrapée !
```

Table des matières

Parcours de séquence.

```
1  def parcours ( objet ) -> None :
2      for k in range(len(objet)) :
3          print(objet[k], end=" - ")
4      print()
5
6  chaine : str = "Choucroute"
7  print("Affichage de la chaine")
8  print(chaine)
9  print("Appel de parcours() : ")
10 parcours(chaine)
11
12
13 liste=list(chaine)
14 print("Affichage de la liste")
15 print(liste)
16 print("Appel de parcours() : ")
17 parcours(liste)
18 .
19 .
```

```
1  >>>
2  Affichage de la chaine
3  Choucroute
4  Appel de parcours() :
5  C - h - o - u - c - r - o - u - t - e -
6
7
8  Affichage de la liste
9  ['C', 'h', 'o', 'u', 'c', 'r', 'o', 'u', 't',
10 Appel de parcours() :
11 C - h - o - u - c - r - o - u - t - e -
```

♡Les chaînes, les listes, les tuples sont des séquences itérables.♡

Parcours de dictionnaire

```
1  dico={  1:'C' ,
2          2 :'h' ,
3          3:'o' ,
4          4 :'u'}
5  print("Affichage du dictionnaire")
6  print(dico)
7  print("Appel de parcours() : ")
8  parcours(dico)
9  .
10 .
```

```
1  >>> %Run Ex_parcours_sequence.py
2  Affichage du dictionnaire
3  {1: 'C', 2: 'h', 3: 'o', 4: 'u'}
4  Appel de parcours() :
5  Traceback (most recent call last):
6    File "/home/Ex_parcours_sequence.py",
7     line 26, in <module>
8      parcours(dico)
9    File "/home/Ex_parcours_sequence.py",
10    line 3, in parcours
11     print(objet[k], end=" - ")
12 KeyError: 0
```

Mauvaise manière de parcourir le dictionnaire, ce n'est pas une séquence !

Parcours de dictionnaire - vu en CM3

```python
def parcours_dico_IN ( dico :dict ) -> None :
    for cle in dico :
        print(f"cle={cle}, valeur= {dico[cle]} ")
    print()


dico={ 1:'C' , 2 :'h' , 3:'o' , 4 :'u'}
print("Affichage du dictionnaire")
print(dico)
print("Appel de parcours() : ")
parcours_dico_IN(dico)
```

```
>>> %Run Ex_parcours.py
Affichage du dictionnaire
{1: 'C',
 2: 'h',
  3: 'o',
  4: 'u'}
Appel de parcours() :
cle=1, valeur= C
cle=2, valeur= h
cle=3, valeur= o
cle=4, valeur= u
```

♡Un dictionnaire est itérable, mais n'est pas une séquence.

Parcours d'un ensemble - vu en CM3

```python
def parcours_ENS ( ens : set ) -> None :
    for element in ens :
        print(f"element : {element} ")
    print()


ensemble : set  =  {'N','E','Z'}
print("Affichage de l ensemble :")
print(ensemble)
print("Appel de parcours() : ")
parcours_ENS(ensemble)
```

```
Affichage de l ensemble :
{'N', 'Z', 'E'}
Appel de parcours() :
element : N
element : Z
element : E
```

♡Un ensemble est juste itérable. Ce n'est pas une séquence.

Synthèse - qui est qui ?

♡Synthèse :

Type	Itérable	Séquence	Indexable
list	oui	oui	oui
tuple	oui	oui	oui
str	oui	oui	oui
dict	oui	non	non
set	oui	non	non

Table des matières

Vérification de l'appartenance.

```
1  chaine : str ="choucroute"
2
3  liste : list =['a','b','c']
4
5  couple : tuple =( 3 , 14 )
6
7  dico : dict ={ 'Léo':'vélo' ,
8              'Léa':'Zumba'}
9
10 ens : set = {23,27,47}
```

```
1  >>> "c" in chaine
2  True
3  >>> "z" in liste
4  False
5  >>> 3  in  couple
6  True
7  >>>  "Léo" in dico
8  True
9  >>> "vélo" in dico
10 False
11 >>>   23 in ens
12 True
```

La commande `element in structure` est un booléen, True si l'élément est dans la structure et False sinon. Très pratique, évite d'écrire le code pour parcourir la structure et tester n fois.

Vérification de l'appartenance pour un dictionnaire.

```
dico : dict ={ 'Léo':'vélo' ,
        'Léa':'Zumba'}

# extraction des data
listeCle : list   = dico.keys()
listeValeur :list = dico.values()

c='Léa'
if c in listeCle) :
    print(f"{c} pratique : {dico[c]}")

val='vélo'
if val in listeValeur :
    print(f"{val} est une activité.")
```

```
>>>
Léa pratique : Zumba
vélo est une activité.
```

Table des matières

Fonction avec mutable.

```
P : list = list(range(10,25,5))
S = P # copie des étiquettes
      # et non des valeurs !!
print(f"P= {P}")
print(f"S= {S}")
S[0]=0
print()
print(f"P= {P}")
print(f"S= {S}")
print("P (mutable) a changé aussi")
print()

def modif_liste( L:list ) -> None :
    L[0]=999

print("modif_liste(P)")
modif_liste( P )
print(f"P= {P}")
print(f"S= {S}")
print("P et S (mutables) ont été changés "
      "par la procédure.")
```

- Rappel :La copie simple de liste ne copie que les adresses. Une modification de l'une modifie l'autre. Les listes sont mutables.
- La fonction est ici une procédure (pas de return). Lors de l'appel de la fonction, L sera une copie simple de la future liste.
- la liste L a les mêmes adresses que S et P, donc les changements sur L se font aussi sur P et S. La procédure convient bien. "return L" est inutile !

Fonction avec mutable.

```python
P : list = list(range(10,25,5))
S = P # copie des étiquettes et non des valeurs !!
print(f"P= {P}")
print(f"S= {S}")
S[0]=0
print()
print(f"P= {P}")
print(f"S= {S}")
print("P (mutable) a changé aussi")
print()

def modif_liste( L ) -> None :
    L[0]=999

print("modif_liste(P)")
modif_liste( P )

print(f"P= {P}")
print(f"S= {S}")
print("P et S (mutables) ont été changés "
      "par la procédure.")
```

```
>>> %Run EX_mutabilite_light.py
P= [10, 15, 20]
S= [10, 15, 20]

P= [0, 15, 20]
S= [0, 15, 20]
P (mutable) a changé aussi

modif_liste(P)
P= [999, 15, 20]
S= [999, 15, 20]
P et S (mutables) ont été changés
par la procédure.
```

Fonction avec mutable.

Même comportement avec un dictionnaire, ensemble.Avec les structures de données mutables.

Voir EX 4 du TPN°3 :

```
1 def ajouteBoite( medicament : str , nb :int, dico : dict) -> None :
2 """Augmente le stock pour un médicament donné"""
```

Dans le TP N°3 , uniquement des procédures ont été écrites pour modifier le stock de médicament (qui était modélisé par un dictionnaire !)

Factorisation de code

```python
print("Aires de différentes formes")

# Aire du rectangle R1
a = 5
b = 10
aire = a * b
print("Aire de R1:", aire)

# Aire du rectangle R2
a2 = 7
b2 = 3
aire2 = a2 * b2
print("Aire de R2:", aire2 )

# Aire du cercle C1
r = 4
aire3 = 3.14159 * r * r
print("Aire de C1:", aire3)
```

```python
# Aire du cercle C2
r2 = 6
aire4 = 3.14159 * r2 * r2
print("Aire de C2:", aire4)

# Aire du triangle T1
base = 8
hauteur = 5
aire5 = (base * hauteur) / 2
print("Aire de T1:", aire5)

# Aire du triangle T2
base2 = 6
hauteur2 = 4
aire6 = (base2 * hauteur2) / 2
print("Aire de T2:", aire6)
```

Ce code est difficile à lire. Répérons les motifs répétitifs.

Factorisation de code

```python
def aire_rectangle(lon, lar):
    return lon * lar

def aire_cercle(r):
    return 3.14159 * r * r

def aire_triangle(b, h):
    return (b * h) / 2

formes = {
    "R1": ("rectangle", 5, 10),
    "R2": ("rectangle", 7, 3),
    "C1": ("cercle", 4),
    "C2": ("cercle", 6),
    "T1": ("triangle", 8, 5),
    "T2": ("triangle", 6, 4),
}
```

```python
print("Aires de différentes formes")

# Liste des clés (nom des formes)
noms = formes.keys()

for nom in noms:
    d = formes[nom]

    if d[0] == "rectangle":
        aire = aire_rectangle(d[1], d[2])
    elif d[0] == "cercle":
        aire = aire_cercle(d[1])
    elif d[0] == "triangle":
        aire = aire_triangle(d[1], d[2])

    print(f"Aire de {nom}: {aire:.2f}")
```

La définition des fonctions permet d'éviter les répétitions inutiles. Ce code est parfaitement lisible. Pour ajouter des formes il suffit uniquement de modifier le dictionnaire formes.

Table des matières

Module python exemple

En TP 1 et TP2 on a vu le contenu du fichier **module_EX_3_TP1.py**

```python
def est_un_chiffre(c : str) -> bool :

def  est_un_chiffre_v2(c : str) -> bool :

def  est_un_chiffre_v3(c : str) -> bool :

def est_un_entier_positif(chaineInput:str)-> bool :

def demander_entier(question : str) -> int:

# Test pour l'utiliser en direct ou l'importer ailleurs
if __name__ == "__main__":
      # S'exécutera seulement si le fichier est exécuté directement
    NbTotalEleve : int = demander_entier("Nombre total d'élèves : ")
    TailleEquipe : int = demander_entier("Nombre de joueurs par équipe : ")
    print(f"Le nombre total d'élèves : {NbTotalEleve}")
    print(f"La taille d'une équipe :  {TailleEquipe}")
```

Module python exemple

En TP2 on a importé une fonction créée précédemment.

```
1 from module_EX_3_TP1 import demander_entier as entrer_entier
```

Cette instruction va chercher dans le module : **module_EX_3_TP1** la fonction **demander_entier** et la renomme en **entrer_entier**.

Avantage : permet de développer des fonctions utilisables dans plusieurs autres codes.

A pratiquer en TP...

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