# B73 Scripting sous Linux

avec python™

## Fonctions, modules, et librairie standard

# Les fonctions

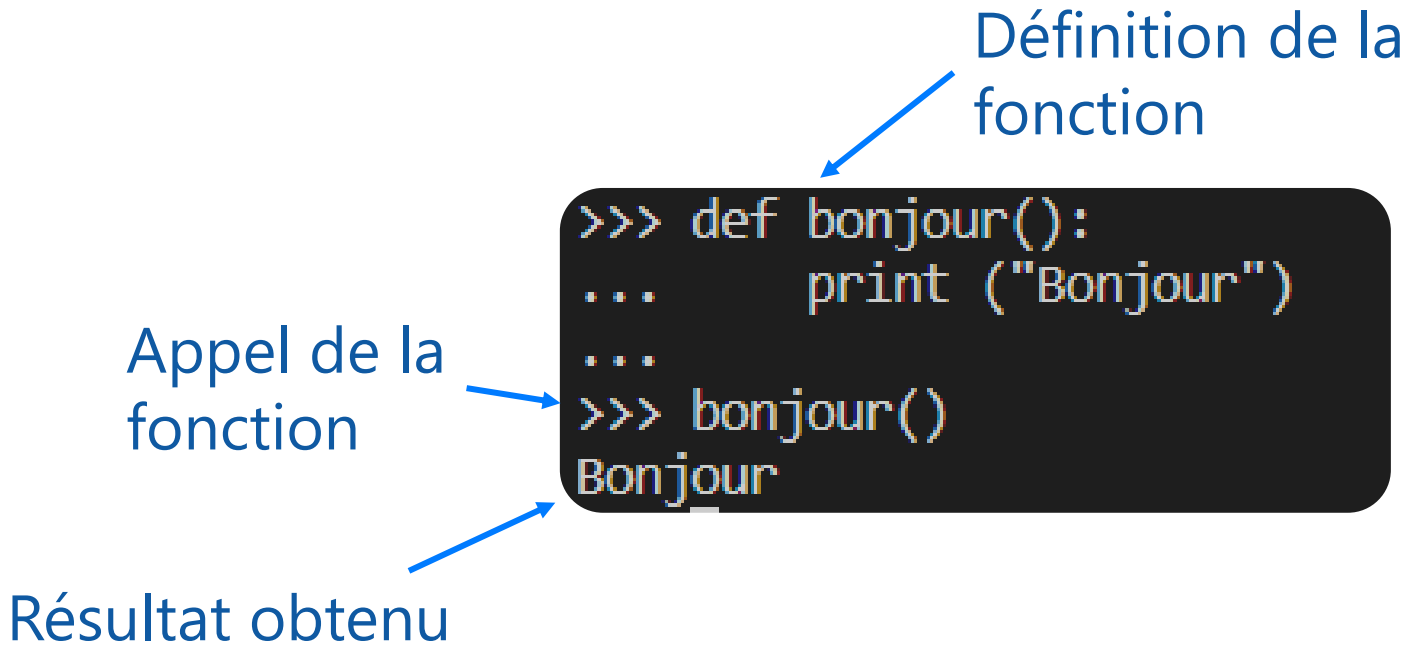
# Les fonctions

```
1   def message_invitation(nom):
2       print(f"""\nBonjour {nom} et bienvenue au cours de programmation 2.
3                 \nProgrammation objet pour profil résautique\n""")
4
```

- Blocs de code réutilisable.
- Permet de structurer et d'organiser le code en séparant les tâches en sous-tâches plus simples.
- Rend le code plus lisible en donnant des noms significatifs aux différentes tâches.
- Peuvent prendre des paramètres d'entrée (ou pas) et renvoyer des résultats (ou pas). Permet de les utiliser de manière flexible dans différentes parties du programme.
- Rend le code plus maintenable.

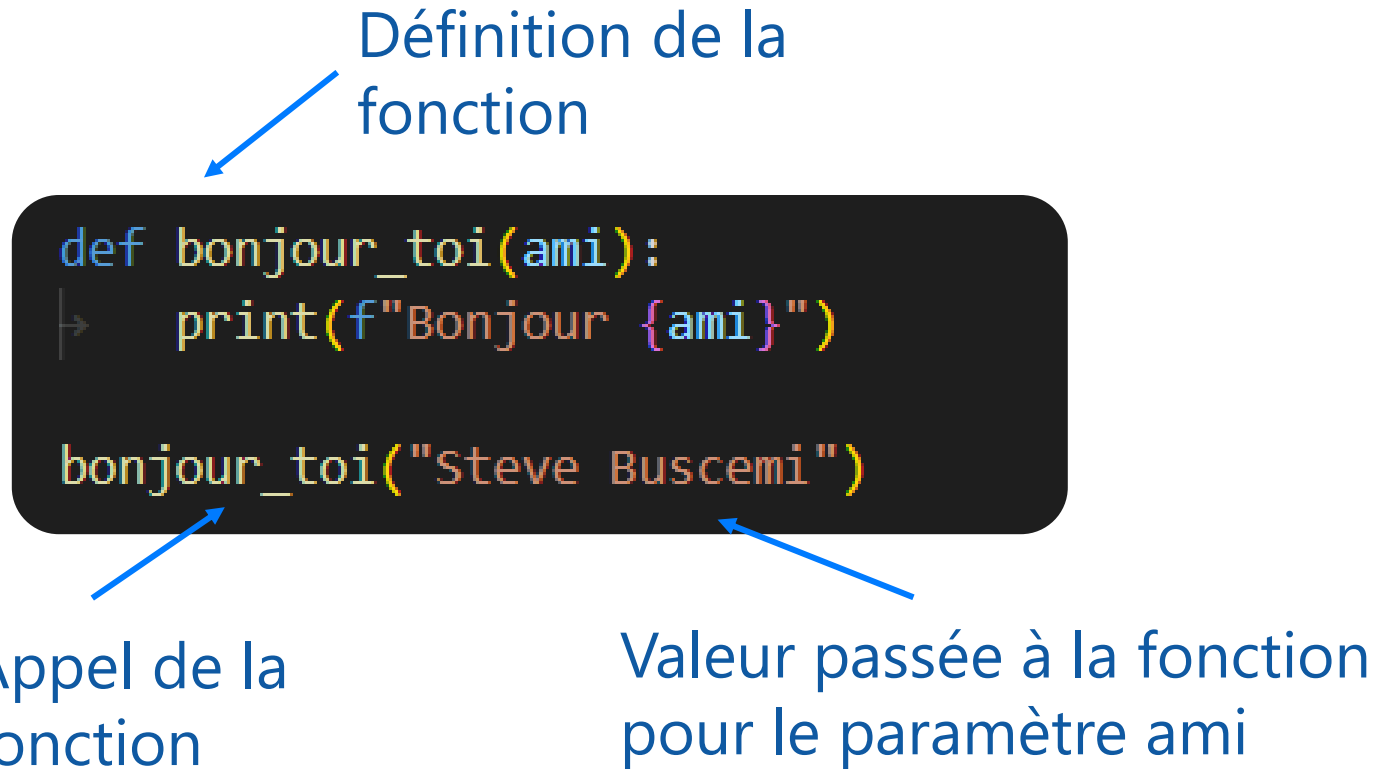
# Fonctions sans paramètres

- Toutes les fonctions commencent avec le mot-clef « def » suivie du nom de la fonction et des parenthèses qui prennent ou non des arguments.
- Cette fonction est sans paramètre.



# Fonctions avec paramètres

- Les fonctions avec paramètres reçoivent une valeur pour chaque paramètre lors de l'appel.



# Fonctions avec paramètres

- Les fonctions avec paramètres peuvent comporter des valeurs par défaut. La variable prendra cette valeur si aucune n'est fournie lors de l'appel de la fonction.

Valeur par défaut pour le paramètre ami.

Appel de la fonction avec et sans valeur donnée pour ami.

```
def bonjour_toi(ami="mon ami"):
    print(f"Bonjour {ami}.")

bonjour_toi("Steve Buscemi")
bonjour_toi()
```

✓ 0.5s

```
Bonjour Steve Buscemi.
Bonjour mon ami.
```

# Fonctions avec valeur de retour

- Quand une fonction retourne une valeur (avec return), il faut capturer cette valeur dans une variable lors de l'appel.

- Le retour du premier appel n'a pas été capturé. Le message produit n'est pas utilisable ailleurs dans le code.

```
20  def bonjour_toi(ami="mon ami"):
21      return(f"Bonjour {ami}.")
22
23  bonjour_toi("Steve Buscemi")
24
25  a = bonjour_toi()
26  print(a)
27
```

PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL

```
R07 Exercices Solution/R07_ex1_Fonctions.py"
Bonjour mon ami.
```

- C'est comme faire un appel à quelqu'un et ne pas écouter sa réponse.

  (ou demander une question au prof et ne pas écouter la réponse)

# Les fonctions

- Peuvent agir différemment selon les valeurs passées.
- Les fonctions peuvent appeler d'autres fonctions, encore une fois pour éviter les répétitions, et faciliter la lisibilité et maintenance du code.

exemples.py > ...

```python
def impression_liste (texte="Liste : ",liste=None):
    if liste == None:
        print("Oops ! utilisé cette fonction avec une liste")
    else:
        print(texte)
        for valeur in liste:
            print(f"ID: {id(valeur)} Cours: {valeur}")
        print("")
```

exemples.py > impression_liste_et_id > liste

```python
def impression_liste_et_id (liste):
    for valeur in liste:
        print(f"ID: {id(valeur)} | {valeur}")


def impression_liste_avec_titre(texte="Liste : ",liste=None):
    if liste == None:
        print("Oops ! utilisé cette fonction avec une liste")
    else:
        print(texte)
        impression_liste_et_id(liste)
        print("")
```

# Modules

- Scripts de code Python réutilisables dans différents programmes.
- Regroupent fonctions, variables et autres éléments ayant une fonctionnalité similaire.
- Code mieux organisé et plus facile à maintenir.
- Différentes parties du code séparées en modules distincts.
- Python possède la librairie standard qui regroupe de nombreux modules prédéfinis et prêts à l'emploi. Donc le code est déjà écrit et testé.

# Modules

- Il y a plusieurs façons de faire l'import, les deux premières sont équivalentes et très bonnes.

```
/* test.py
1   import subprocess
2   import subprocess as sb
3   from subprocess import run
4   from subprocess import *
5
6   sb.run(["echo","test sb"],shell=True)
7
```

- On peut renommer les modules pour simplifier notre code, surtout si les modules ont des noms long ou complexe.

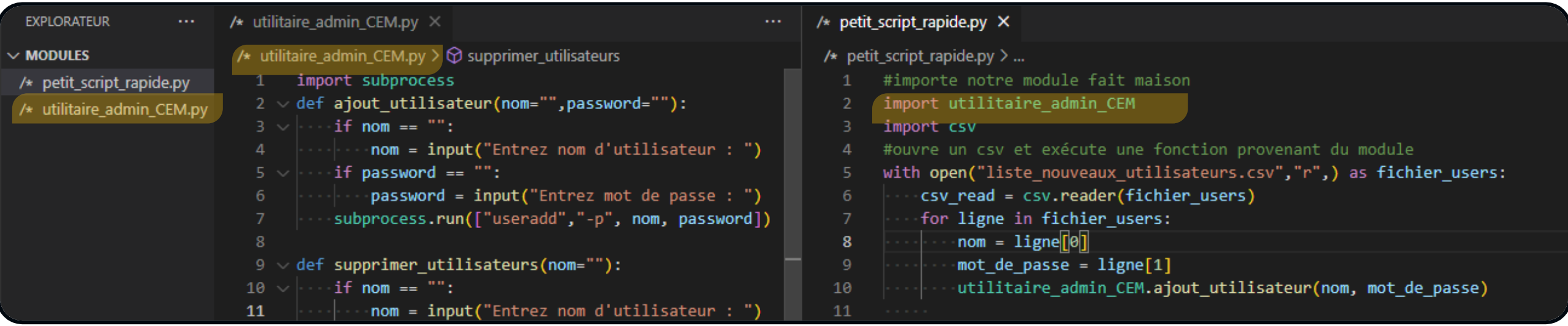
- Les syntaxes aux lignes 3 et 4 sont valides mais peuvent causer des conflits.

# Modules

- Utiliser des modules est très facile. On les utilise déjà dans le cours.

```
import os
utilisateur = os.getenv(“USER”)
```

- Créer de nouveaux modules est aussi simple que d’écrire un script



# Range / scope des variables

- Quand une variable est définie globalement elle est disponible dans le script et dans les fonctions de ce script.
- Mais quand une variable est redéfinie localement,  la variable globale ne change pas. Cela fait juste une autre variable qui est locale à la fonction.
- Lorsqu'on définie des variables dans une fonction, elles sont locales à cette fonction. Ce qui signifie qu'elles n'existent que dans la fonction.
- Lorsqu'on termine l'exécution d'une fonction, les variables locales à cette fonction cessent d'exister.

# Le module os

- environ : dictionnaire des variables de système
  - Getenv() et putenv() pour obtenir ou modifier ces variables
- os.Chdir() : change le répertoire dans lequel l'interprétateur « agit »
- os.Getcwd() : affiche le répertoire courant
- os.mkdir()  : crée un répertoire
- os.rmdir() : supprime un répertoire VIDE
- os.remove() : supprime un fichier
- os.rename : renomme un fichier ou répertoire
- os.path() : sous-module pour travailler avec les paths
- os.syteme() : exécute des commandes dans le shell

# Le module sys

- sys.argv : contient les valeurs passées en arguments
  - (souvent utilisé avec le module argparse)
- sys.exit() : permet de sortir d'un script de façon « propre » et en envoyant un message de retour
- sys.platform : indique la platforme dans laquelle le script est exécuté (si on veut des parties différentes exécutées dans linux et window)
- sys.path : liste des répertoires où l'interprétateur recherche les modules

# Le module subprocess

- subprocess.run(*arg, shell=True)
  - Permet de passer des commandes au shell à partir du code python.
  - Très utile pour la gestion de systèmes

- Subprocess.Popen : Permet de lancer de nouveaux processus et d'interagir avec.