RÉSUMÉ THÉORIQUE - FILIÈRE INFRASTRUCTURE DIGITALE

M106 - Automatiser les tâches d'administration

42 heures

SOMMAIRE

MODALITÉS PÉDAGOGIQUES

1

LE GUIDE DE SOUTIEN

Il contient le résumé théorique et le manuel des travaux pratiques

2

LA VERSION PDF

Une version PDF est mise en ligne sur l'espace apprenant et formateur de la plateforme WebForce Life

3

DES CONTENUS TÉLÉCHARGEABLES

Les fiches de résumés ou des exercices sont téléchargeables sur WebForce Life

4

DU CONTENU INTERACTIF

Vous disposez de contenus interactifs sous forme d'exercices et de cours à utiliser sur WebForce Life

5

DES RESSOURCES EN LIGNES

Les ressources sont consultables en synchrone et en asynchrone pour s'adapter au rythme de l'apprentissage

PARTIE 1

Découvrir les concepts de base de la programmation

Dans ce module, vous allez :

- Découvrir la programmation structurée
- Découvrir la programmation orientée objet
- Utiliser les conditions et les boucles

17 heures

CHAPITRE 1

Découvrir les concepts de base de la programmation

Ce que vous allez apprendre dans ce chapitre :

- Découvrir la méthode de résolution d'un problème
- Découvrir les algorithmes
- Découvrir les structures de base du langage Python

05 heures

CHAPITRE 1

Découvrir la programmation structurée

1. **Description de la méthode de résolution d'un problème : raisonnement par algorithme**
2. Description d'un algorithme
3. Les structures de base du langage Python

Raisonnement par algorithme

Pour faire exécuter une tâche par ordinateur :

- Il faut, tout d'abord, détailler suffisamment les étapes de résolution du problème pour qu'elles soient exécutables par l'homme.
- Ensuite, transférer la résolution en **une suite d'étapes élémentaires et simples à exécuter**, pouvant être codées en **un programme** dans un langage compréhensible par ordinateur. Toute **suite d'étapes élémentaire et simple à exécuter** s'appelle un ALGORITHME.

Pour pouvoir écrire des algorithmes, il faut connaitre la résolution manuelle du problème, connaitre les capacités de l'ordinateur en terme d'actions élémentaires qu'il peut assurer et la logique d'exécution des instructions.

Pour résoudre un problème, un algorithme effectue plusieurs tâches :

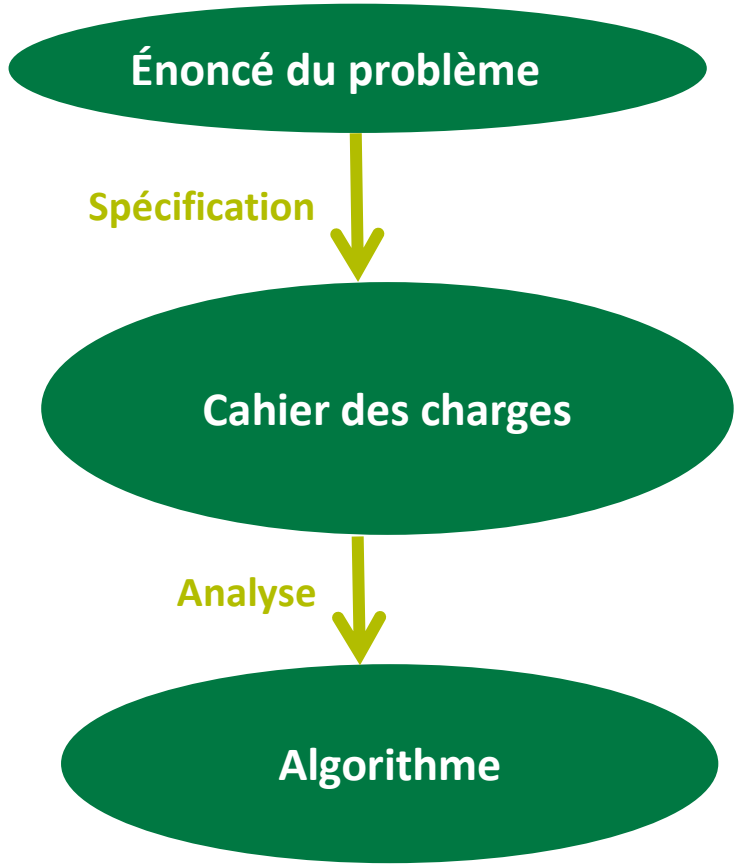
- Saisie et contrôle de données
- Affichage
- Recherche
- Calcul
- ...

Étapes de résolution d'un problème

1. Comprendre l'énoncé du problème ;
2. Décomposer le problème en sous-problèmes plus simples à résoudre ;
3. Associer à chaque sous-problème, une spécification :
    - Les données nécessaires
    - Les données résultantes

La démarche à suivre pour arriver au résultat en partant d'un ensemble de données.

4. Élaboration d'un **algorithme**.



Phase d'écriture de l'algorithme

- C'est une phase de traduction des phases précédentes en se basant sur les règles du langage algorithmique.
- Le langage **algorithmique** est un langage proche du langage humain et qui est défini par un ensemble de mots réservés et de syntaxes d'écriture d'actions.

La suite de ce chapitre présente l'algorithmique, l'algorithme et les structures de base du langage Python permettant d'exécuter les algorithmes.

Algorithmique

- La réalisation de programmes passe par l'écriture d'algorithmes.
    - D'où l'intérêt de l'**Algorithmique**.
- Le terme **algorithme** vient du nom du mathématicien arabe **Al-Khawarizmi** (820 après J.C.).
- Un algorithme est une description complète et détaillée des actions à effectuer et de leur séquencement pour arriver à un résultat donné.
    - Intérêt : séparation analyse/codage (pas de préoccupation de syntaxe).
    - Qualités : **exact** (fournit le résultat souhaité), **efficace** (temps d'exécution, mémoire occupée), **clair** (compréhensible), **général** (traite le plus grand nombre de cas possibles), …
- **L'algorithmique** désigne la discipline qui étudie les algorithmes et leurs applications en informatique.

Une bonne connaissance de l'algorithmique permet d'écrire des **algorithmes** exacts et efficaces.

CHAPITRE 1

Découvrir la programmation structurée

1. Description de la méthode de résolution d'un problème : raisonnement par algorithme
2. **Description d'un algorithme**
3. Les structures de base du langage Python

Définitions :

- Algorithme : ensemble des étapes permettant d'atteindre un but en répétant un nombre fini de fois un nombre fini d'instructions ;
- Pour faire exécuter une tâche par ordinateur, il faut tout d'abord, détailler suffisamment les étapes de résolution du problème, pour qu'elles soient exécutable par l'homme. Ensuite, transférer la résolution en **une suite d'étapes élémentaires et simples à exécuter**, pouvant être codée en **un programme** dans un langage compréhensible par ordinateur. Toute **suite d'étapes élémentaires et simples à exécuter** s'appelle un ALGORITHME ;
- **Un algorithme** décrit une succession d'opérations qui, si elles sont fidèlement exécutées, produiront le résultat désiré ;
- **Un algorithme** est une suite d'actions que devra effectuer un automate pour arriver, en un temps fini, à un résultat déterminé à partir d'une situation donnée. La suite d'opérations sera composée d'actions élémentaires appelées **instructions ;**
- C'est la logique d'écrire des algorithmes. Pour pouvoir écrire des algorithmes, il faut connaitre la résolution manuelle du problème, connaitre les capacités de l'ordinateur en terme d'actions élémentaires qu'il peut assurer et la logique d'exécution des instructions.

Représentation d'un algorithme

- Historiquement, il y a deux façons de représenter un algorithme :
- **L'organigramme :** représentation graphique avec des symboles (carrés, losanges, etc.).
    - Offre une vue d'ensemble de l'algorithme
    - Représentation quasiment abandonnée aujourd'hui.
- **Le pseudo-code :** représentation textuelle avec une série de conventions ressemblant à un langage de programmation (sans les problèmes de syntaxe).
    - Plus pratique pour écrire un algorithme
    - Représentation largement utilisée

Objets algorithmiques

- Un algorithme manipule des objets qui représentent des données ou des résultats. Ces objets peuvent être des constantes ou des variables (valeur modifiables).
- Un objet est caractérisé par :
    - Un identificateur : un moyen permettant d'identifier l'objet de façon unique au niveau de la mémoire de l'ordinateur ;
    - Une nature : constante ou variable ;
    - Un type : caractérisant l'ensemble des valeurs possibles (son domaine de définition) ainsi que les opérateurs applicables.

Identificateur

- Une suite de caractères alphanumériques (sans espace) commençant obligatoirement par une lettre.

Exemple :

- Identificateurs valables : note, ma_note, moyenne_1, moyenne2.
- Identificateurs non valables : 1moyenne, mot?, chaine, demi-mot, deux notes.

Constante

- Un objet dont la valeur reste fixe durant l'exécution d'un algorithme ;
- Caractérisée par un nom qui est son identificateur ;
- Initialisée à la déclaration ;
- Par convention : noter les constantes en majuscules.

Exemple : MAX=1000

Variable

- Un objet dont le contenu peut varier tout au long de l'algorithme.

Exemple : moy: réel

	Opérateur	Algorithmique	PYTHON
Opérateurs Arithmétiques	Addition	+	+
	Soustraction	-	-
	Multiplication	*	*
	Puissance	^	**
	Division	/	/
	Division entière	Div	//
	Reste de la division entière	Mod	%
Opérateurs Relationnels	strictement inférieur	x < y	x < y
	strictement supérieur	x > y	x > y
	Inférieur ou égal	x <= y	x <= y
	Supérieur ou égal	x >= y	x>=y
	Égalité	x = y	x == y
	Différence	x < > y	x != y

Opérateurs

Table de vérité :

	Opérateur	Algorithmique	PYTHON
Opérateurs logiques	Conjonction	Et	and
	Disjonction	Ou	or
	Négation	Non	not

X	Y	NON(X)	X ET Y	X OU Y
V	V	F	V	V
V	F	F	F	V
F	V	V	F	V
F	F	V	F	F

PARTIE 1

Types élémentaires

Un type permet d'indiquer l'ensemble des valeurs possibles ainsi que les opérations que l'on peut effectuer sur ce type.

Exemple : entier, réel.

1. Types élémentaires : entier

Sous-ensemble des entiers relatifs.

Domaine de définition	Sous-ensemble de **Z** **-32768 ... 32767**
Opérateurs arithmétiques	+,-,*, /, DIV, MOD
Opérateurs relationnels	<, >, =, ≤, ≥, ≠

2. Types élémentaires : réel

Sous-ensemble des nombres réels.

Domaine de définition	Sous-ensemble de **Z** **-32768 ... 32767**
Opérateurs arithmétiques	+,-,*, /
Opérateurs relationnels	<, >, =, ≤, ≥, ≠

3. Types élémentaires : booléen

Vrai/Faux.

Domaine de définition	Vrai, faux
Opérateurs logiques	Non (négation) Et (conjonction) Ou (disjonction) Ouex (ou exclusif)

Types élémentaires

4. Types élémentaires : caractère

Chiffres, lettres (majuscules et minuscules) et les symboles spéciaux.

Domaine de définition	• Lettres alphabétiques minuscules et majuscule • Chiffres de 0 à 9 • Symboles spéciaux
Opérateurs relationnels	<, >, =, ≤, ≥, ≠

5. Types élémentaires : chaine de caractères

- Succession de *n* caractères, *n* étant compris entre 0 et 255. Si *n* est nul on parle d'une chaine vide ;
- Les caractères relatifs à une chaine sont représentés entre guillemets ;
- On peut accéder au ième caractère d'une chaine *ch* en utilisant la notation *ch[i]* avec $1 \leq i \leq Long\ (ch)$.

Structure d'un algorithme

ALGORITHME nom_de_l'algorithme

CONST

{Définition des constantes}

TYPE

{Définition de types}

VAR

{Déclaration de variables}

DEBUT

{Corps de l'algorithme: suite d'instructions}

FIN

Structure simple

Une structure est dite simple (appelée encore une séquence) si elle ne contient que des instructions :

- d'entrée de données ;
- de sortie de résultats ;
- d'affectation.
- **Affectation :** l'opération d'affectation permet d'affecter une valeur à une variable. Cette valeur peut être une autre variable, le résultat retourné par une fonction, le résultat d'une expression logique ou arithmétique ou encore une constante.

Syntaxe : variable ← expression

Exemple :

X ← 15.5

Y ← X

Z ← X * Y

Structure simple

- **Opération d'entrée :** l'instruction Lire permet à l'utilisateur d'introduire des valeurs pour qu'elles soient utilisées par le programme.

Syntaxe : Lire(variable)

- **Opération de sortie :** l'instruction Ecrire permet d'afficher un message ou un résultat à l'utilisateur.

Exemple : Ecrire(«Bonsoir»)

Ecrire(moy)

Ecrire(« la moyenne est » moy)

Structure conditionnelle simple	Structure conditionnelle complète	Structure conditionnelle imbriquée
Si *condition* **Alors** Traitement **FinSi**	**Si** condition **Alors** Traitement1 **Sinon** Traitement2 **FinSi**	**Si** condition1 **Alors** Traitement 1 **Sinon** **Si** condition2 **Alors** Traitement2 **Sinon** Traitement3 **FinSi** **FinSi**

Bilan

- La programmation structurée permet de formaliser un algorithme en utilisant exclusivement les 4 familles d'instructions pour la définition du flux d'exécution :
    1. Gestion des variables ;
    2. Entrées-sorties ;
    3. Conditions ;
    4. Boucles.
- Un problème peut être décomposé en sous-tâches qui peuvent :
    - être paramétrées par la fourniture de valeurs par la tâche appelante, qui sont stockées dans des variables appelées paramètres.
    - retourner une valeur à la tâche appelante.
- Une sous-tâche peut également afficher un résultat au lieu de retourner une valeur. On parle alors de procédure, tandis qu'une sous-tâche qui renvoie une valeur est communément appelée une fonction.
- Lorsque l'utilisateur saisit des valeurs, il est important de :
    1. Lui indiquer ce qui est attendu par un affichage.
    2. Gérer les erreurs potentielles de saisie.

Nous finalisons ce chapitre par la **présentation des structures de base du langage Python** permettant d'exécuter les algorithmes.

CHAPITRE 1

Découvrir la programmation structurée

1. Description de la méthode de résolution d'un problème : raisonnement par algorithme
2. Description d'un algorithme
3. **Les structures de base du langage Python**

Structure conditionnelle simple	Structure conditionnelle complète	Structure conditionnelle imbriquée
if *condition* Bloc d'instructions	**if** condition Bloc d'instructions 1 **else** Bloc d'instructions 2	**if** condition1 Bloc d'instructions 1 **else** **if** condition2 Bloc d'instructions 2 **else** Bloc d'instructions 3

La structure itérative Tant que

- Exécution d'un traitement donné en boucle tant qu'une condition spécifiée est vraie (True).

- Ce type de structure est utilisé chaque fois que le nombre d'itérations qui seront exécutées n'est pas connu à l'avance.

	Algorithmique	PYTHON
Syntaxe	**Tant que** Condition **Faire** Traitement **Fin Tant que**	**while** condition: ..... Bloc d'instructions .....
Exemple	I ⟵ 2 Tant que i<=20 Faire Ecrire(i*2) Fin Tant que	i=0 while i<=5: print(i) i+=1

La structure itérative Répéter jusqu'à

- Ce type de structure permet l'exécution d'un traitement donné plusieurs fois jusqu'à ce qu'une condition soit vraie.
- Ce type de structure est utilisée à chaque fois que le nombre d'itérations n'est pas connu à l'avance.

	Algorithmique	PYTHON
Syntaxe	**Répéter** Traitement **Jusqu'à** Condition	La boucle Répéter... jusqu'à n'existe pas en PYTHON mais on peut la simuler en utilisant la boucle while et l'instruction break.
Exemple	i ⟵ 2 Répéter Ecrire(i*2) i ⟵ i + 1 Jusqu'à (i>20)	

PARTIE 1

En algorithmique

Syntaxe :

Pour compteur de Valeur_initiale à Valeur_finale **Faire**

Traitement

Fin Pour

Description :

- Cette structure itérative permet de répéter un traitement donné un certain nombre de fois connu à l'avance.
- Le compteur (variable de contrôle) prend la valeur initiale (Valeur_initiale) au moment d'accès à la boucle puis, à chaque parcours, il passe **automatiquement** à la valeur suivante dans son domaine jusqu'à atteindre la valeur finale (Valeur_finale).

Exemple :

Pour i de 1 à 5 **Faire**

Ecrire(i)

Fin Pour

En PYTHON

Description :

- La boucle Pour est traduite en PYTHON avec la boucle **for** (avec la fonction **range**).
- La fonction range est employée avec un, deux ou trois paramètres.

Exemple 1 :

```
somme=0
for x in range(1, 10):
    somme=somme+x
print("La somme des 10 premiers entiers est", somme)
```

Exemple 2 :

```
for x in range(3):
    print(x)
```

Exemple 3 :

```
for i in range(2,10,3):
    print(i)
```

Description

- L'instruction **break** dans une structure itérative permet l'arrêt immédiat de la boucle (for ou while) dans laquelle se trouve l'instruction ;
- L'instruction break a presque le même effet que l'utilisation d'une variable booléenne pour sortir d'une boucle while ;
- L'instruction break rend possible l'arrêt d'une boucle for. Ceci n'est pas recommandé ! Si vous avez un traitement itératif qui doit s'arrêter si une condition est vérifiée, il est plus adéquat d'utiliser une boucle while.

Exemple

```
For i in range(10)
                print("debut iteration",  i)
                If i==2:
                        break
                print("fin iteration ",  i)
print(«  apres la boucle »)
```

Output :

```
debut iteration 0
fin iteration  0
debut iteration 1
fin iteration1
debut iteration 2
fin de la boucle
```

Exemple

L'instruction **continue** permet de passer prématurément au tour de boucle suivant.

```
for x in range(5):
    if x==3:
        continue
    print(x)
```

Output :

0

1

2

4

- Site officiel : https://www.python.org/
- Python est un langage de programmation développé depuis 1989 par le développeur néerlandais Guido Van Rossum et de nombreux collaborateurs.
- PYTHON est diffusé en 1991.
- Licence libre.

PYTHON

Description :

- La boucle Pour est traduite en PYTHON avec la boucle **for** (avec la fonction **range**).
- La fonction range est employée avec un, deux ou trois paramètres.

Exemple 1 :

```
somme=0
for  x in  range(1, 10):
    somme=somme+x
print("La somme des 10 premiers entiers est", somme)
```

Exemple 2 :

```
for x in range(3):
    print(x)
```

Exemple 3 :

```
for i in range(2,10,3):
    print(i)
```

Caractéristiques du langage Python

- **Haut niveau** : un programme Python est 3 à 5 fois moins cours qu'un programme C ;
- **Orienté objet** : supporte l'héritage et la surcharge des opérateurs ;
- **Libre** : **« free»** utilisation sans restriction dans les projets commerciaux ;
- **Portable** : peut fonctionner sur différentes plateformes OS (operating system) ;
- **Interprété.**
- **Dynamiquement typé** : tout objet manipulable par le programmeur possède un type bien défini à l'exécution qui n'a pas besoin d'être déclaré à l'avance ;
- **Efficace** : Python intègre un système de gestion d'**exceptions** permettant de simplifier considérablement la gestion des erreurs ;
- **Doté d'un gestionnaire de mémoire** : Python gère ses ressources (mémoire, descripteurs de fichiers...) sans intervention du programmeur.
- **Facile** à apprendre/à lire/à écrire/à maintenir (syntaxe très simple et combinée à des types de données évolués (listes, dictionnaires, etc.).
- **Évolutif** : Python est un langage qui continue à évoluer grâce à une communauté d'utilisateurs très actifs.
- **Puissant** : doté de bibliothèques très variées (numpy, scipy, etc.).

Que peut-on faire avec PYTHON ?

- Python est un langage de programmation de plus en plus populaire.
- Domaines d'application :
    - Des graphiques (matplotlib) ;
    - De la bio-informatique (Biopython) ;
    - Data science (pandas, scikit-learn) ;
    - Des applications web ;
    - Interfacer des systèmes de gestion de bases de données ;
    - Du calcul scientifique/statistique (numpy, scipy) ;
    - Des jeux ;
    - Etc.

Bilan

- Un code Python s'écrit dans un fichier avec l'extension .py. Il est exécuté :
  - Soit avec la commande `python nomfichier.py`
  - Soit en tant que script en exécutant le fichier s'il débute par un shebang tel que `#! /usr/bin/env python`
- Python fournit des structures pour les 4 familles d'instructions nécessaires à la programmation structurée :
  - La gestion des variables ;
  - Les entrées/sorties ;
  - Les conditions ;
  - Les boucles.

Le chapitre suivant traite d'une évolution de la programmation structurée et de sa mise en œuvre en Python : la programmation orientée objet.

CHAPITRE 2

Découvrir la programmation orientée objet

Ce que vous allez apprendre dans ce chapitre :

- Découvrir la programmation orientée objet
- Maitriser les notions de classe et d’objet
- Maitriser les notions d'attribut et de méthode
- Maitriser la syntaxe du langage Python

07 heures

CHAPITRE 2

Découvrir la programmation orientée objet

Principe

- Rassembler les données et leurs traitements associés au sein d'une même entité.
- Qu'est-ce qu'on peut modéliser ?
    - Une personne, un livre ;
    - Un point, un vecteur, un polynôme ;
    - Un cercle, un carré, un rectangle ;
    - Un atome, etc.

- Nous allons nous intéresser à ces notions relatives à l'approche objet :
    - Classe ;
    - Attributs, méthodes ;
    - Encapsulation ;
    - Objet, instance ;
    - Instanciation ;
    - Héritage.

Classe

- Dans la description d'un problème réel, il est impossible de décrire tous les objets du domaine. Les classes sont un moyen pour créer des objets qui présentent une structure commune (objets ayant les mêmes caractéristiques) ;
- Une classe est un **modèle** qui décrit plusieurs objets ;
- Les caractéristiques de la classe s'appellent des **attributs** ;
- Les actions qu'on peut appliquer sur un objet sont appelées des **méthodes** ;
- On définit les attributs et les méthodes au niveau de la classe.

CHAPITRE 2

Découvrir la programmation orientée objet

Classe

Exemple :

- Définition d'une classe **Personne** avec comme attributs **nom** et **prenom**.
- modifier_nom, modifier_prenom, retourner_nom et retourner_prenom peuvent être définies comme des méthodes de la classe **Personne**.

Remarques

- Evidemment, on aurait pu retenir d'autres caractéristiques (attributs) et d'autres méthodes pour la classe Personne.

Objet

- La relation qui existe entre une classe et un objet est une relation **d'instanciation** : un objet est une **instance** (occurrence) de classe.
- Des instances (objets) possibles de la classe Personne :
  - Personne1(nom='Ben Yaghlen',prenom='Salma)
  - Personne2(nom='Mokadi',prenom='Ibrahim')

Quand on instancie (crée) des objets, on définit des valeurs aux attributs.

PYTHON est un langage orienté objet :

- Tout est objet : données, fonctions, modules, etc.
- Un objet PYTHON possède :
- Un type (nature de l'objet) : la fonction type permet de retourner le type d'un objet (**classe**).
- Une valeur (des données matérialisées par des valeurs d'attributs).
- Des opérations applicables (**méthodes**).

PARTIE 1

PARTIE 1

Exemple

- Tous les types qu'on a manipulés (int, oat, str, complex, list, tuple, etc.) sont en réalité définis en tant que classes.

Instanciation d'un objet de type complex :

```
> x=1+5j ou bien x=complex(1,5)
> type(x)
        <class 'complex'>
```

Attributs de la classe complex :

```
> x.real
      1.0
> x.imag
      5.0
```

Méthode de la classe complex :

```
> x.conjugate()
      (1-5j)
```

Définition de classe

```
class Personne :
        def __init__( self ) :
        #constructeur de la classe Personne
        #avec parametres par defaut
                self.nom='Ben Yaghlen '
                self.prenom=' Salma ‘
```

- **class** est un mot-clé permettant de définir la classe ;
- Personne est le nom de la classe (par convention, commence par une lettre majuscule) ;
- def __init__(self) est une méthode spéciale appelée **constructeur**, automatiquement appelée lors de l'instanciation d'un objet ;
- Au niveau du constructeur, nous avons défini deux attributs nom et prénom dont les valeurs seront attribuées au moment de l'instanciation (ici des valeurs par défaut) ;
- Le paramètre self (convention) représente l'instance elle-même, il doit apparaitre comme premier paramètre dans toutes les méthodes. Mais, on ne le précise pas lors de l'appel.

PARTIE 1

Exemple

#Instanciation -- Creation de deux objets de la classe

Personne

P1=Personne( )

P2=Personne( )

- P1=Personne(): permet de créer une première instance de la classe Personne.

  ➡ Appel au constructeur __init__ de la classe Personne.

- On ne précise pas le paramètre self du constructeur lors de l'instanciation.

```
print (P1.nom, P1.prenom)
print (P2.nom, P2.prenom)
```

- Les deux instances ont les mêmes valeurs Ben Yaghlen Salma (les valeurs par défaut affectées aux attributs au niveau du constructeur).
- Les attributs sont accessibles en lecture/écriture. Mais il n'est pas conseillé de faire ça en orienté objet. On accédera aux attributs via des méthodes.

```
class Personne :
    def __init__( s e l f , a , b ) :
    #version 2 du constructeur de la classe
    #Personne avec parametres
        self.nom=a
        self.prenom=b
#Instanciation --Creation de deux objets de la classe Personne
P1=Personne ('Ben Slimen ' , 'Marwa ' )
P2=Personne ( ' Gabsi ' , ' Farah ' )
```

Exemple

```
class Personne :
        def __init__( s e l f , a , b ) :
                self.nom=a
                self.prenom=b
        def afficher (self) :
                print ( 'Nom: ' , self. nom)
                print ( 'Prenom : ' , self.prenom)
        def modifier_nom (self, a ) :
                self.nom=a
        def modifier_prenom (self , b ) :
                self.prenom=b
        def retourner_nom (self) :
                return self.nom
        def retourner_prenom (self) :
                return self.prenom
```

Intérêts

- Une application a besoin de services dont une partie seulement est proposée par une classe déjà définie (on ne possède pas nécessairement le code source de cette classe) ;
- Permet de ne pas réécrire tout le code ;
- L'héritage implique :
    - Une relation de généralisation : regroupement au sein d'une classe des caractéristiques communes à un ensemble de classes ;
    - Une relation de spécialisation : ajout de caractéristiques propres à une sous-classe;
- La généralisation et la spécialisation génèrent une hiérarchie de classes.

Exemple

Classe Point :

- A une position ;
- Peut être déplacé ;
- Peut calculer sa position.

Une application a besoin de manipuler des points (comme le permet la classe Point), mais en plus, de dessiner des points à l'écran.

PointGraphique = Point + une couleur + une opération d'affichage

⇒ Une solution simple en POO : l'héritage.

⇒ Définir une nouvelle classe à partir de la classe déjà existante.

Exemple

- Définition de deux sous-classes : classe Etudiant et classe Enseignant qui héritent de tous les attributs et méthodes de la classe Personne.
- Ajout des attributs (num et matière) comme attributs spécifiques ainsi que des méthodes spécifiques à chaque sous-classe.

```
class Etudiant ( Personne ) :
        def __init__( self , a , b , number ) :
                Personne . __init__( self, a , b )
                self.num=number
        def afficher ( self ) :
                Personne . afficher ( self )
                print ( 'Num: '+str ( self . num) )
        def modifier_num ( self, number ) :
                self .num=number
        def retourner_num ( self ) :
                return ( self.num)
```

Exemple

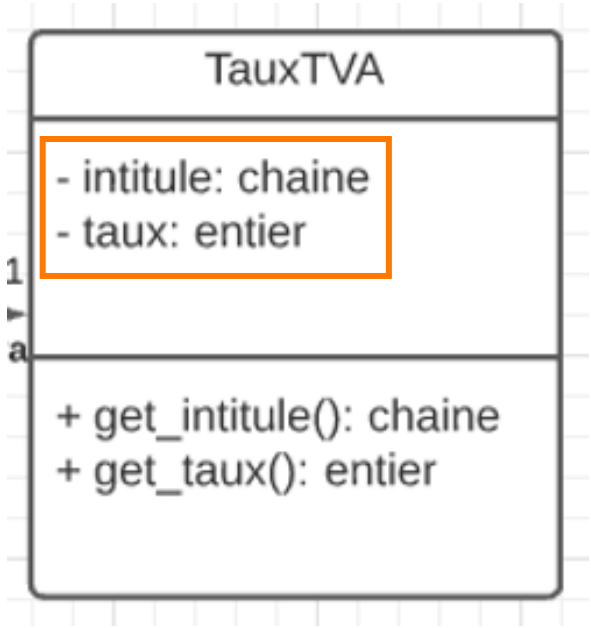
```
class Enseignant ( Personne ) :
        def __init__( self , a , b , mat ) :
                Personne . __init__( self , a , b )
                self . matiere=mat
        def afficher ( self ) :
                Personne . afficher ( self )
                print ( ' Matiere : ' , self . matiere )
        def modifier_mat ( self , mat ) :
                self.matiere=mat
        def retourne r_mat ( self ) :
                return ( self . matiere )
```

CHAPITRE 2

Découvrir la programmation orientée objet

Implémentation des caractéristiques avec les attributs

- Une classe définit des caractéristiques et le comportement d'un objet.
- Les caractéristiques sont matérialisées par des **attributs**, qui sont des **variables** rattachées à la classe, sans lesquelles elle n'aurait pas de sens.
- Par exemple, un téléphone sans écran ni connexion réseau n'est pas fonctionnellement un téléphone !
- Comme pour les variables, un attribut est défini par :
    - Un nom
    - Un type
- Un attribut possède un élément supplémentaire : sa **visibilité**.
    - Un attribut **privé** n'est visible qu'à l'intérieur de la classe ;
    - Un attribut **protégé** n'est visible que dans la classe dans lequel il est défini ou une de ses classes filles ;
    - Un attribut **public** est accessible depuis l'extérieur.
- En modélisation UML, on spécifie la visibilité avec :
    - Un + pour un attribut public ;
    - Un # pour un attribut protégé ;
    - Un – pour un attribut privé, comme dans l'illustration ci-contre.



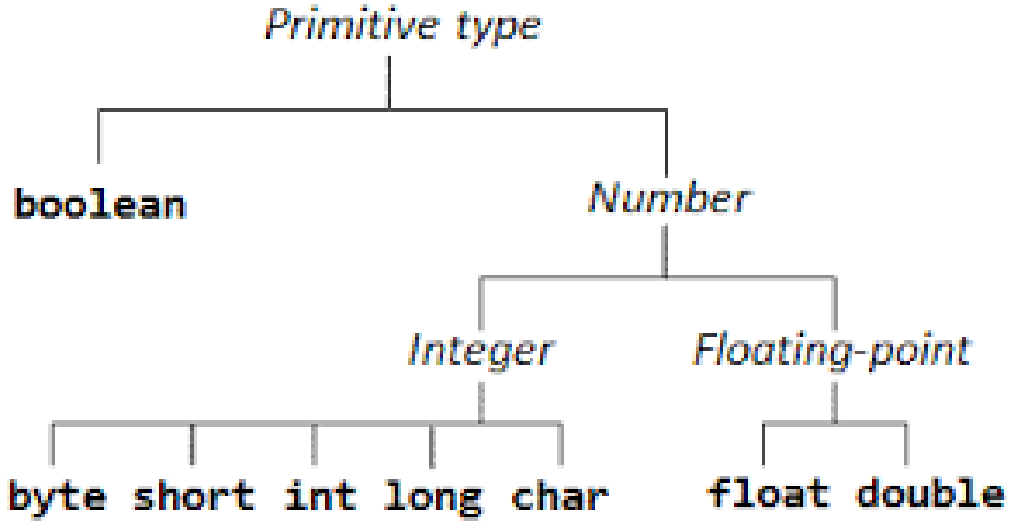
PARTIE 1

Type des attributs

Le type d'un attribut peut être :

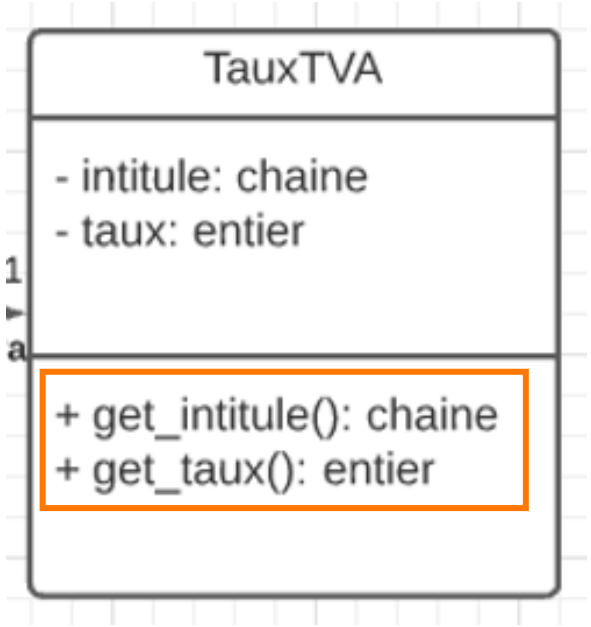
1. Un type primitif comme un entier, un réel, un nombre complexe, un caractère, un booléen ;
2. Une classe prédéfinie dans le langage utilisé. Par exemple, la plupart des langages fournissent une classe **Date** avec un comportement permettant d'effectuer des opérations courantes, comme la comparaison ou la récupération de la date du jour ;
3. Une classe définie par un développeur dans le cadre du programme ou importée depuis une librairie ou un autre programme.

- L'illustration ci-contre présente les types primitifs généralement implémentés dans les différents langages objet.
- La chaine de caractère est un cas particulier : selon les langages, elle sera considérée comme type primitif ou classe. En algorithmie, nous utiliserons le terme **chaine** pour indiquer le stockage de plusieurs caractères représentant un mot ou une phrase.



Implémentation des caractéristiques avec les attributs

- Une classe définit des caractéristiques et le comportement d'un objet.
- Le comportement est formalisé par des **méthodes** qui sont des **fonctions** rattachées à la classe.
- Comme pour les fonctions, une méthode est définie par :
  - Un nom ;
  - 0, 1 ou plusieurs paramètres ;
  - 0 ou 1 valeur de retour.
- Comme les attributs, les méthodes possèdent une **visibilité**.
  - Une méthode **privée** n'est accessible qu'à l'intérieur de la classe ;
  - Une méthode **protégée** n'est accessible que dans la classe dans laquelle elle est définie ou une de ses classes filles ;
  - Une méthode **publique** est accessible depuis l'extérieur.
- Comme pour les attributs, en modélisation UML, on spécifie la visibilité avec :
  - Un + pour une méthode publique, comme dans l'illustration ci-contre.
  - Un # pour une méthode protégée.
  - Un – pour une méthode privée.



PARTIE 1

CHAPITRE 2

Découvrir la programmation orientée objet

Définition de classe en Python

- Les éléments suivants permettent de définir une classe en Python :
  - Le mot-clé **class** suivi du nom de la classe
  - La liste des attributs avec leur type :
    - _ indique un attribut protégé.
    - __ indique un attribut privé.
  - Les méthodes, qui débutent par le mot-clé **def** :
    - La méthode **__init__** est le constructeur. Elle permet d'initialiser les attributs à partir de valeurs par défaut ou définies en paramètres.
    - Par convention, une méthode commençant par *get* est un **accesseur** : elle renvoie la valeur d'un attribut.
    - De même, une méthode commençant par *set* est un *mutateur* : elle permet de modifier la valeur d'un attribut.
    - Les autres méthodes sont les méthodes **métier**.

```python
class Taxe:
    __intitule: str
    __taux: int

    def __init__(self, intitule: str, taux: int) -> None:
        self.__intitule = intitule
        self.__taux = taux

    def get_intitule(self) -> str:
        return self.__intitule

    def get_taux(self) -> int:
        return self.__taux

    def afficher(self) -> None:
        print(f"La taxe {self.__intitule} est au taux de
        {self.__taux}%")
```

Création d'un objet en Python

- Un objet est déclaré comme une variable dont le type est le nom de la classe.
    - Si la classe est créée dans un autre fichier, il faut l'importer avec la syntaxe **from nom_fichier_sans_.py import Classe.**
- La création s'effectue en appelant le nom de la classe suivi des arguments attendus par le constructeur entre parenthèses.
- L'objet créé est assigné à la variable.
- Une méthode est appelée avec la syntaxe **objet.methode(arguments).**
- L'exemple ci-contre illustre :
    - L'import de la classe Taxe depuis le fichier **taxe.py ;**
    - La création de 2 objets depuis la classe **Taxe ;**
    - L'appel de la méthode métier **afficher()** sur chaque objet ;
    - L'affichage d'une chaine personnalisée s'appuyant sur les valeurs récupérées via les accesseurs **get_intitule() et get_taux().**

```
from taxe import Taxe

droit_import: Taxe
tva_normale: Taxe

droit_import = Taxe("droit d'importation", 25)
tva_normale = Taxe("TVA normale", 20)

droit_import.afficher()
tva_normale.afficher()

print(f"Les taxes {droit_import.get_intitule()} et
{tva_normale.get_intitule()} donnent un taux cumulé de
{droit_import.get_taux() + tva_normale.get_taux()}%")
```

Création d'une classe fille en Python

- La création d'une classe fille implique :
  - L'import de la classe mère depuis le fichier la contenant ;
  - Le mot-clé *class* suivi du nom de la classe fille puis, entre parenthèses, de la classe mère ;
  - La déclaration d'attributs complémentaires si nécessaire ;
  - Un constructeur faisant appel au constructeur de la classe mère avec la fonction **super().**
- Si les attributs de la classe mère sont publics ou protégés, ils sont accessibles dans la classe fille. Sinon ils doivent être manipulés via les accesseurs et les mutateurs s'ils existent.
- Les méthodes existant dans la classe mère sont utilisables.
- Si une méthode est redéfinie avec le même nom dans la classe fille, elle remplace celle de la classe mère. Dans notre exemple :
  - La méthode **afficher** est redéfinie ;
  - La méthode **calculer_prix_ttc** est ajoutée ;
  - Les *getters* de la classe mère sont utilisables.

```python
from taxe import Taxe

class TauxTVAHerite(Taxe):

    def __init__(self, intitule: str, taux: int) -> None:
        super().__init__(intitule, taux)

    def calculer_prix_ttc(self, prix_ht: float) -> float:
        return prix_ht + prix_ht * self.get_taux() / 100

    def afficher(self) -> None:
        print(f"La TVA au taux {self.get_intitule()} est de
        {self.get_taux()}%")
```

Bilan de notre découverte programmation orientée objet

- La Programmation Orientée Objet (POO) améliore la programmation pour permettre la création de programmes plus complexes.
- La POO s'appuie sur 5 grands principes : la modularité, la spécialisation, la flexibilité, la décomposition de problème et la réutilisation.
- Une classe est un **modèle** qui définit les caractéristiques et le comportement des objets qu'elle représente.
    - Les caractéristiques d'un objet sont définies par ses **attributs ;**
    - Le comportement d'un objet est défini par ses **méthodes ;**
    - Les attributs et les méthodes ont une visibilité publique, protégée ou privée.
- Python permet la création de classes avec le mot-clé **class**.
- Les objets sont créés à partir de classes et manipulés via les variables qui les contiennent.

À suivre

- Le dernier chapitre de cette partie revient sur l'utilisation de conditions et de boucles grâce aux opérateurs mathématiques et logiques.

CHAPITRE 3

Utiliser les conditions et les boucles

Ce que vous allez apprendre dans ce chapitre :

- Découvrir les opérateurs mathématiques et logiques
- Comprendre le traitement conditionnel et le traitement itératif

05 heures

CHAPITRE 3

Utiliser les conditions et les boucles

1. **Opérateurs mathématiques et logiques**
2. Traitement conditionnel
3. Traitement itératif

Les types d'opérateurs

- En algorithmie comme dans les langages qui l'implémentent, on distingue 5 types d'opérateurs :
    1. Les opérateurs d'affectation, qui permettent d'assigner une valeur à une variable. En Python, il s'agit du =.
    2. Les opérateurs logiques, que nous détaillerons dans la suite de cette section ;
    3. Les opérateurs booléens, qui évaluent un résultat à True ou False ;
    4. Les opérateurs de comparaison, qui correspondent aux opérateurs :
        - < pour strictement inférieur
        - > pour strictement supérieur
        - <= pour inférieur ou égal
        - >= pour supérieur ou égal
        - == pour l'égalité, afin de différencier l'opérateur d'égalité de l'opérateur d'affectation =
        - != pour la différence
    5. Les opérateurs mathématiques, que nous détaillerons dans la suite de cette section.

La suite de cette section s'appuie sur le langage Python pour la présentation des opérateurs.

Les opérateurs mathématiques

- Les opérations mathématiques courantes sont exploitables via les symboles +, -, *, **, /, // et %.
    - Le tableau ci-contre recense l'usage de chaque opérateur.
- D'autres opérations sont disponibles via des fonctions fournies par le module *math* de la librairie standard. Par exemple :
    - math.floor(*x*) renvoie la partie entière du réel x par troncature ;
    - math.sin(*x*) renvoie le sinus d'un x exprimé en radians ;
    - math.pow(*x*, *y*) renvoie la valeur de x puissance y ;
- https://docs.python.org/3/library/math.html# recense l'ensemble des opérations fournies par le module.

symbole	opération	types	exemples
+	Addition	entier, réel chaîne de caractères	6+4 == 10 "a" + "b"== "ab"
-	Soustraction	entier, réel	6-4 == 2
*	Multiplication	entier réel chaîne de caractères	6*4 == 24 1.2 * 1 == 1.2 3 * "s" == "sss"
**	Puissance	entier, réel	12**2 == 144
/	Division	entier réel	6/4 == 1 (Python 2) 1.5 (Python 3) 6./4 == 1.5
//	Division entière	entier, réel	6//4 == 1
%	Modulo	entier, réel	6%4 == 2

PARTIE 1

Opérateurs logiques

- Il existe 4 opérateurs logiques :
    1. Le **OU logique**, exprimé par l'opérateur **|**
    2. Le **ET logique**, exprimé par l'opérateur &
    3. Le **OU** exclusif logique, exprimé par l'opérateur ^
    4. Le **NON** logique, exprimé par l'opérateur ~
- Les opérateurs logiques s'appliquent à chaque bit d'un opérande.

Par exemple :

- 0011 | 0101 == 0111
- 0011 & 0101 == 0001
- 0011 ^ 0101 == 1000
- ~0011 == 1100

Opérateurs booléens

- Les opérateurs booléens reprennent le principe des opérateurs logiques pour renvoyer True (vrai) ou False (faux). Python en définit 3 :
    - **and** renvoie **Vrai** si tous les opérandes s'évaluent à **Vrai**.
    - **Or** renvoie **Vrai** si un des opérandes s'évalue à **Vrai**.
    - **not** renvoie **Vrai** si l'opérateur est **Faux**.
- L'opérande peut être :
    - Un booléen ;
    - Un entier, qui est évalué à False s'il vaut 0, et à True sinon ;
    - Une chaine de caractères, qui est évaluée à True si elle n'est pas vide ;

Par exemple :

- "" and 0 == False
- "" or 0 == False
- "" or 8 == False
- "False" and 8 == True
- Les opérations booléennes sont au cœur des boucles et conditions multiples, que nous traitons dans la suite de ce chapitre.

CHAPITRE 3

Utiliser les conditions et les boucles

1. Opérateurs mathématiques et logiques
2. **Traitement conditionnel**
3. Traitement itératif

Rappel

- Une condition permet l'exécution d'une instruction ou d'une série d'instructions en fonction du résultat d'un test.

Il existe 3 types de conditions :

1. **L'exécution conditionnelle :** un bloc d'instructions n'est exécuté que si le test est positif. Le pseudo-code algorithmique est :

```
SI condition
ALORS instructions exécutées si VRAI
FIN SI
```

2. **L'alternative :** selon le résultat du test, un bloc ou un autre est exécuté.

```
SI condition
ALORS instructions si VRAI
SINON instructions si FAUX
FIN SI
```

3. **Le choix :** plusieurs chemins sont possibles. Si aucun cas n'est validé, des instructions par défaut sont fournies.

```
SELON QUE
cas 1 : instructions si la cas 1 est VRAI
cas 2 : instructions si le cas 2 est VRAI
AUTREMENT : instructions par défaut si aucun cas n'est validé
```

PARTIE 1

Conditions en Python

- Une condition permet l'exécution d'une instruction ou d'une série d'instructions en fonction du résultat d'un test.

Il existe 3 types de conditions :

1. L'exécution conditionnelle en Python s'écrit en indentant le bloc exécuté si **condition** s'évalue à **True**.

```
if condition:
    # instructions si condition s'évalue à Vrai
```

2. L'alternative s'écrit avec la syntaxe if/else.

```
if condition:
    # instructions si condition s'évalue à True
else:
    # instructions si condition s'évalue à False
```

3. Le choix évalue une égalité entre la variable suivant le mot-clé **match** et les différentes valeurs indiquées après chaque *case*. Un cas par défaut est défini par le **case** _
    - Le code ci-contre reprend l'exemple présenté au chapitre 1, où **direction** est une variable pouvant contenir les directions définies par les points cardinaux.

```
direction: str
direction = input("Où souhaitez-vous aller ?")
match direction : # exécution conditionnelle
        case "est":
            print("Vers le soleil levant")
        case "nord":
            print ("Vers le haut")
        case "ouest":
            print("Vers le soleil couchant")
        case "sud":
            print ("Vers le bas")
        case _: # cas par défaut (autrement)
            print("Vous êtes perdu ?")
```

Utilisation des opérateurs booléens pour des conditions multiples

- Une condition peut nécessiter l'évaluation de plusieurs opérandes pour s'exécuter.

  Par exemple :

  - L'appel de la fonction **voter()** est conditionnée à la valeur de l'âge **et** de la non privation des droits civiques (par exemple, suite à une condamnation).

```
if age > 18 and droits_civiques == True:
    voter()
```

  - Un paiement peut s'effectuer par plusieurs moyens.

```
if carte_bleue == "acceptée" or contenu_porte_monnaie >= 54:
   payer (54)
```

  - Un traitement peut s'exécuter si une variable de garde n'est pas positionnée.

```
if not fini:
```

- Si plus de deux opérandes sont évalués avec des opérateurs logiques, l'ordre de priorité s'applique : **not** puis **and** puis **or**.
  - Pour forcer l'exécution d'un **or** avant un **and**, il faut utiliser des parenthèses.

  Par exemple :

```
cond1 and (cond2 or cond3)
```

- Nous finalisons ce chapitre avec l'utilisation des boucles.

CHAPITRE 3

Utiliser les conditions et les boucles

Rappel

- Une itération – ou boucle – permet de répéter une instruction ou une série d'instructions jusqu'à ce qu'une **condition d'arrêt** soit atteinte.

On distingue 2 types de boucles :

1. **La boucle bornée :** elle s'exécute un nombre de fois prédéfini, en utilisant un compteur.

```
Compteur : nombre entier
POUR  compteur VARIANT de 0 à 10
instructions
FIN POUR
```

2. **La boucle non bornée :** elle s'exécute tant qu'une condition renvoie VRAI.

```
maximum : nombre entier
Valeur : nombre entier
maximum <- 20
valeur <- 0
TANT QUE valeur < maximum // au début 0 < 20, donc la boucle s'éxécute
instructions
FIN TANT QUE // si valeur n'atteint jamais 20, la boucle est infinie !
```

Boucles en Python

1. La boucle bornée s'effectue avec la syntaxe **for… in**.
    - **in** est suivi d'un **itérateur**, c'est-à-dire d'une fonction ou d'une structure sur laquelle on peut itérer ;
    - Dans le chapitre 1 nous avons introduit la fonction **range**, qui génère les nombres compris entre les valeurs fournies en argument ;
    - Les structures complexes que nous aborderons en partie 2 sont également utilisables pour les itérations bornées.
2. La boucle non bornée s'appuie sur l'opérateur **while** suivi d'une **condition d'arrêt**. Celle-ci peut être simple, comme dans l'exemple du chapitre 1, ou composé de plusieurs opérandes évaluées grâce aux opérateurs booléens.
    - L'exemple ci-contre illustre une boucle non bornée : la boucle s'arrête dès que le tirage au sort génère un 6.
    - Comme illustré ci-dessous, 2 exécutions successives exécutent la boucle de manière différente.

```
Tu as tiré 4, encore 9 essais
Tu as tiré 1, encore 8 essais
Tu as tiré 5, encore 7 essais
Tu as tiré 1, encore 6 essais
Tu as tiré 6, c'est gagné
```

```
Tu as tiré 1, encore 9 essais
Tu as tiré 6, c'est gagné
```

```python
import random

tirage: int = 0
limite: int = 10
essais: int = 0
fini: bool = False

while essais < limite and not fini:
    essais += 1
    tirage = random.randint(1,6)
    if tirage == 6:
        fini = True
        print(f"Tu as tiré {tirage}, c'est gagné")
    else:
        print(f"Tu as tiré {tirage}, encore {limite - essais} essais")

if tirage != 6:
    print("C'est perdu")
```

PARTIE 1

Bilan de l'utilisation des conditions et des boucles

- Il existe 5 types d'opérateurs : d'affectation, logiques, booléens, de comparaison, mathématiques.
- Les opérateurs permettent de créer des combinaisons pour utiliser des conditions multiples et des itérations s'appuyant sur plusieurs conditions.
- Python permet d'implémenter les 3 types de conditions :
    - Exécution conditionnelle avec **if**
    - Alternative avec **if… else**
    - Choix avec **match... Case**
- Nous pouvons également coder les 2 types de boucles :
    - Bornées avec **for... In**
    - Non bornées avec **while**

À suivre

- La partie suivante traite du développement de programmes. Nous allons commencer par la conception de programmes dans l'environnement de développement, puis nous verrons comment créer un script pour faciliter les opérations de gestion.

PARTIE 2

Développer des programmes

Dans ce module, vous allez :

- Concevoir des programmes
- Créer un script pour faciliter les opérations de gestion

09 heures

CHAPITRE 1

Concevoir des programmes

Ce que vous allez apprendre dans ce chapitre :

- Installation et découverte des outils de programmation
- Ecriture des programmes dans l'environnement de développement
- Exécution des programmes dans l'environnement de développement

05 heures

CHAPITRE 1

Concevoir des programmes

1. **Installation et découverte des outils de programmation**
2. Ecriture des programmes dans l'environnement de développement
3. Exécution des programmes dans l'environnement de développement

PYTHON : installation

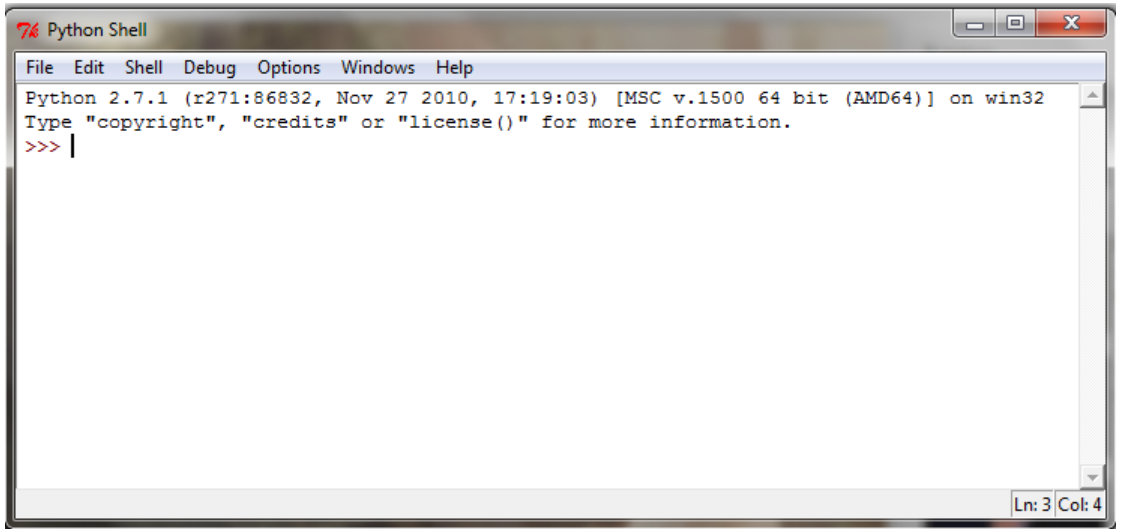
- PYTHON offre un environnement de développement intégré (IDE : Integrated Development Environment) : IDLE.
- **Sous Windows :** pour installer Python avec l'**environnement de développement IDLE**, il suffit de le télécharger puis de l'installer.
- Une fois installé, vous pouvez lancer IDLE en allant dans :

Démarrer → Programmes → Python → IDLE (Python GUI)

- Python peut être utilisé en deux modes : **mode interactif** ou en **mode script**.

1. Mode interactif

- Les instructions tapées sont exécutées directement par l'interpréteur Python.



Opérations élémentaires

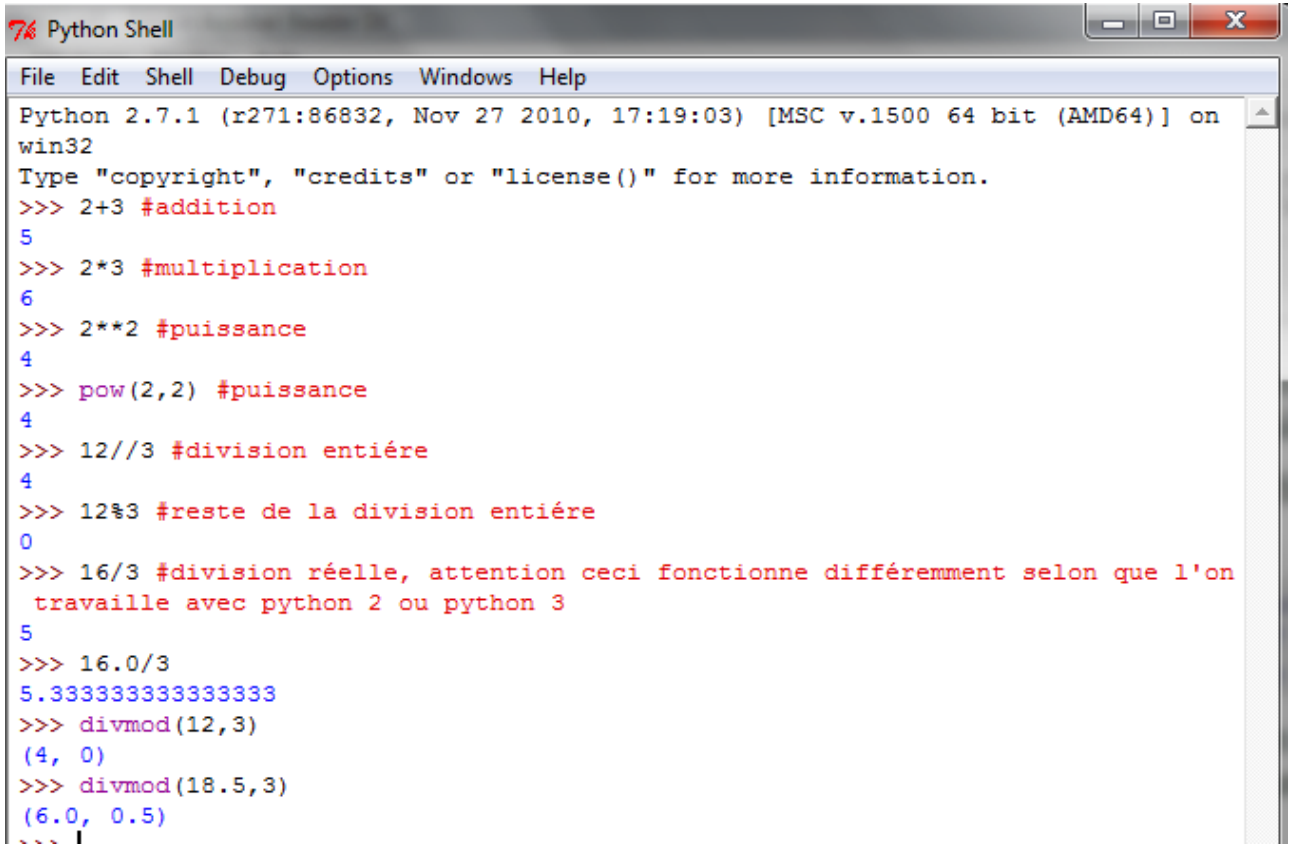
Exemple :

```
> > > 2 + 2
4
```

> > > prompt ou invite de commande.

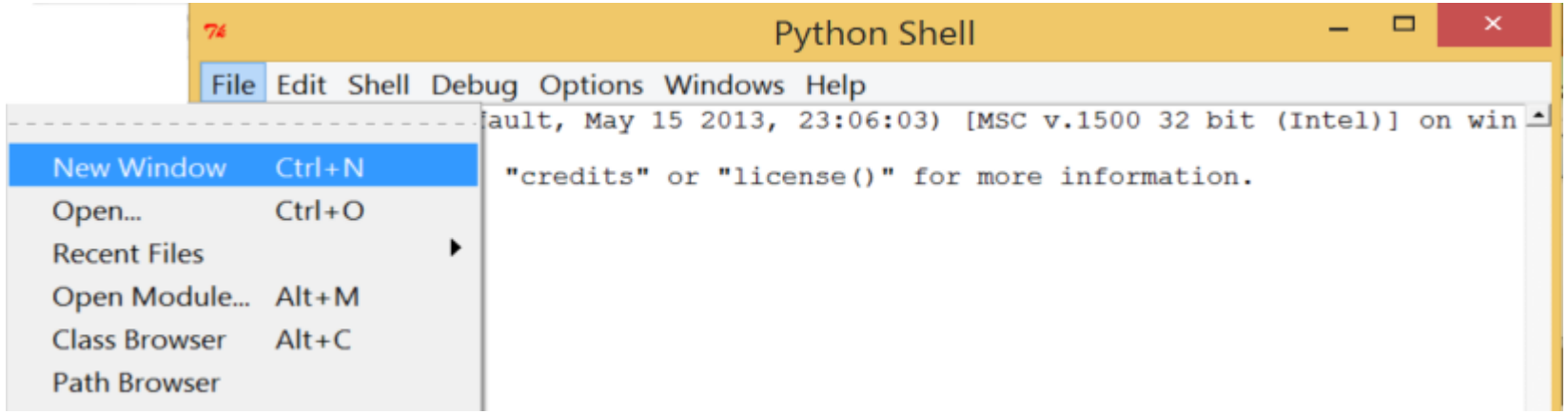
2 + 2 : instruction PYTHON, entrée par l'utilisateur.

4 : réponse de l'interprète PYTHON.

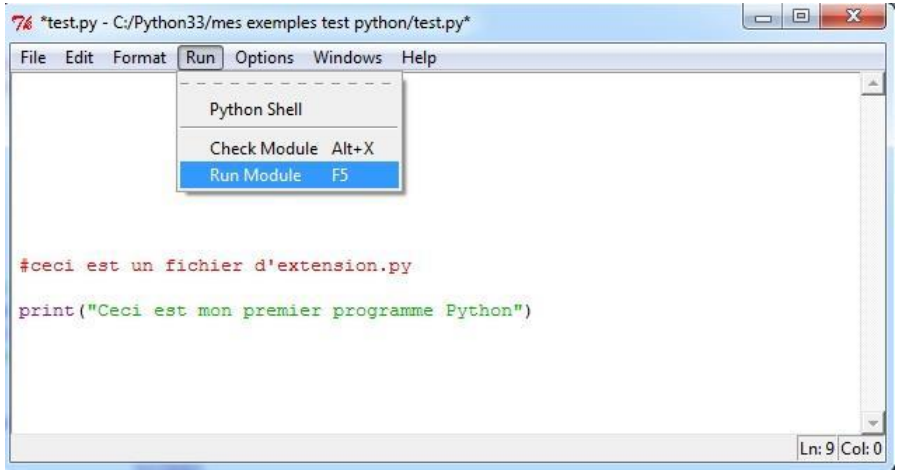


PARTIE 2

2. Mode script



Une nouvelle fenêtre s'ouvre, vous écrivez votre code Python et enregistrez le tout dans un fichier d'extension .py



PARTIE 2

2. Mode script

modescript.py - C:/Users/sousou/Desktop/livre python/modescr...

File Edit Format Run Options Windows Help

```
#ceci est un commentaire
#ceci est un fichier source python pouvant contenir tout type d'instruction
#il faut écrire le code, enregistrer le fichier source d'extention .py
#l'éxécution se fait par la touche F5
print('Ceci est mon premier programme Python')
```

Ln: 3 Col: 23

Exécution :

Python Shell

File Edit Shell Debug Options Windows Help

```
Python 3.2.5 (default, May 15 2013, 23:06:03) [MSC v.1500 32 bit (Intel)] on win
32
Type "copyright", "credits" or "license()" for more information.
>>> ========== RESTART ==========
>>>
Ceci est mon premier programme Python
>>>
```

PARTIE 2

Bilan

- Un shell est un interpréteur de commandes interactif permettant d'interagir avec l'ordinateur. On utilisera le shell pour lancer l'interpréteur Python.
- Python peut être utilisé en deux modes : **mode interactif** ou en **mode script**.
- La partie suivante traite du développement de programmes en Python.

CHAPITRE 1

Concevoir des programmes

1. Installation et découverte des outils de programmation
2. **Ecriture des programmes dans l'environnement de développement**
3. Exécution des programmes dans l'environnement de développement

Introduction

- Python est un **langage orienté objet.**
  - Tout est objet : données, fonctions, modules...
- Un objet **B** possède :
  - Une identité **id** (adresse mémoire), **id(B)**.
  - Un type, **type(B)** : intrinsèque (int, float, str, bool...) ou défini par l'utilisateur à l'aide d'une classe.
  - Des données (valeur).
- L'opérateur **is** compare l'identité de 2 objets (adresses).
- L'opérateur **==** compare le contenu de deux objets.

On n'a pas besoin de spécifier le type d'un objet qu'on manipule. PYTHON gère ça automatiquement. On parle d'un **typage dynamique**.

Les types élémentaires en Python

- **int** : les nombres entiers
- **bool** : True, False
- **str** : les chaines de caractéres
- **float** : les nombres réels
- **complex** : les nombres complexes
- **NoneType** : seule valeur possible est None. C'est la valeur retournée par une fonction qui ne retourne rien.

- **1re manière**

```
>>>import math
>>>dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos',……., 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>> math.ceil(7.8989) #partie entière supérieur
8
>>> math.floor(7.8989) #partie entière inférieur
7
>>>help(math.ceil)
Help on built-in function ceil in module math:
ceil(...)
ceil(x)
Return the ceiling of x as an int.
This is the smallest integral value >= x.
```

- **2e manière : utilisation d'un alias**

```
>>>import math as m
>>>dir(m)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos',……., 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>> m.sqrt(2)
1.4142135623730951
```

- **3e manière : importation de toutes les fonctions d'un module**

```
>>>from math import *
>>>sqrt(2)
1.4142135623730951
```

Écrire un script

- Un script Python est un fichier texte dont l'extension est **.py**
- Pour écrire du code dans un script Python : utilisez un éditeur de texte comme *Notepad*. Des éditeurs plus évolués comme *Notepad++* permettent de changer la couleur du texte en fonction de ce qu'il représente. On utilisera **Spyder** qui est un logiciel dédié à l'écriture du code en Python.

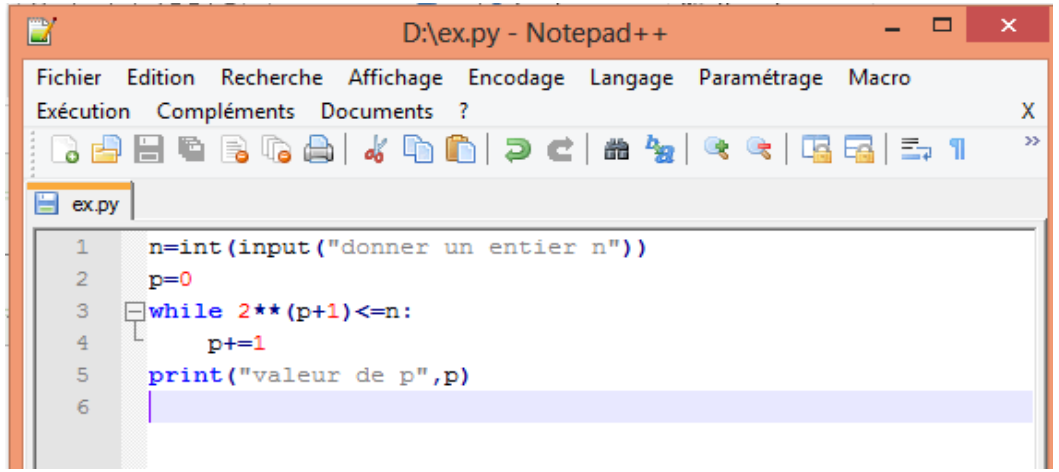
Exemple :

Soit *n* un entier supérieur à 1.

Écrire un script Python qui détermine le seul entier *p*, positif ou nul tel que $2^p <= n < 2^{p+1}$

Exemple :

```
n=int(input("donner un entier n"))
p=0
while 2**(p+1)<=n:
        p+=1
print("valeur de p",p)
```



PARTIE 2

CHAPITRE 1

Concevoir des programmes

1. Installation et découverte des outils de programmation
2. Ecriture des programmes dans l'environnement de développement
3. **Exécution des programmes dans l'environnement de développement**

Exécuter un script depuis Spyder

Depuis Spyder :

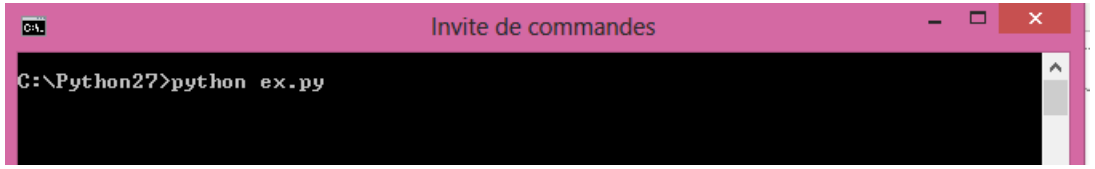
On peut ouvrir un script Python dans **Spyder** puis l'exécuter :

- Ligne par ligne avec la touche **F9**.
- Une sélection spécifique (une partie d'une ligne, plusieurs lignes, etc.) avec la touche **F9**.
- Une cellule encadrée par le symbole **# %%** avec la combinaison **Ctrl + Entrée**.
- Entièrement avec la touche **F5** ou en cliquant sur la flèche verte.

Exécuter un script depuis Python

Depuis Python : exécution en ligne de commande

- Pour exécuter le programme :
  - Sous Windows : ouvrir l'invite de commande (CMD)
  - Sous Mac OS : ouvrir Terminal.
  - Taper la commande suivante et appuyer sur Entrée : **python ex.py**

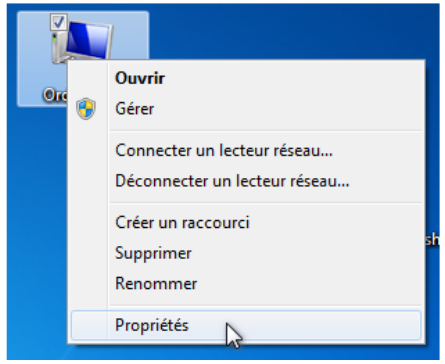


Remarques

- En cas d'erreur indiquant **« Python n'est pas reconnu en tant que commande interne ou externe » :** cette erreur se produit lorsque le chemin vers Python.exe n'est pas défini.

Il faut définir le chemin avant la compilation de la manière suivante :

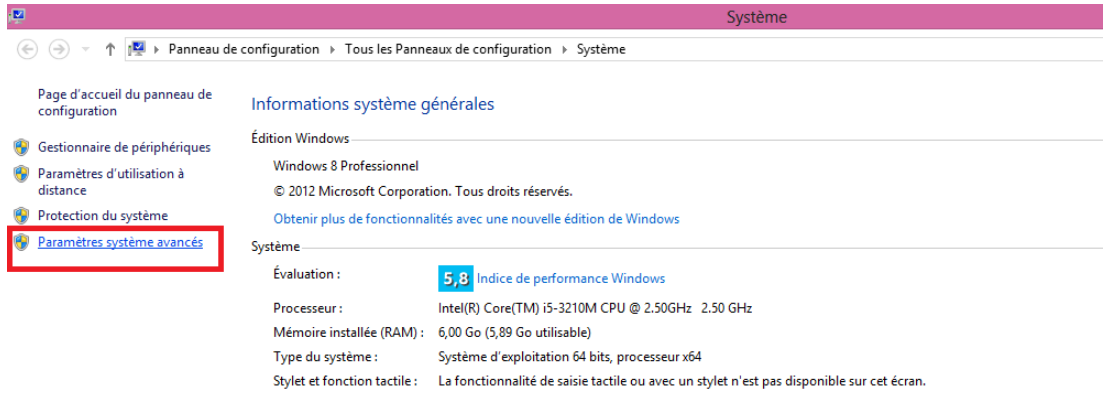
1. **Accéder aux propriétés de votre ordinateur.**



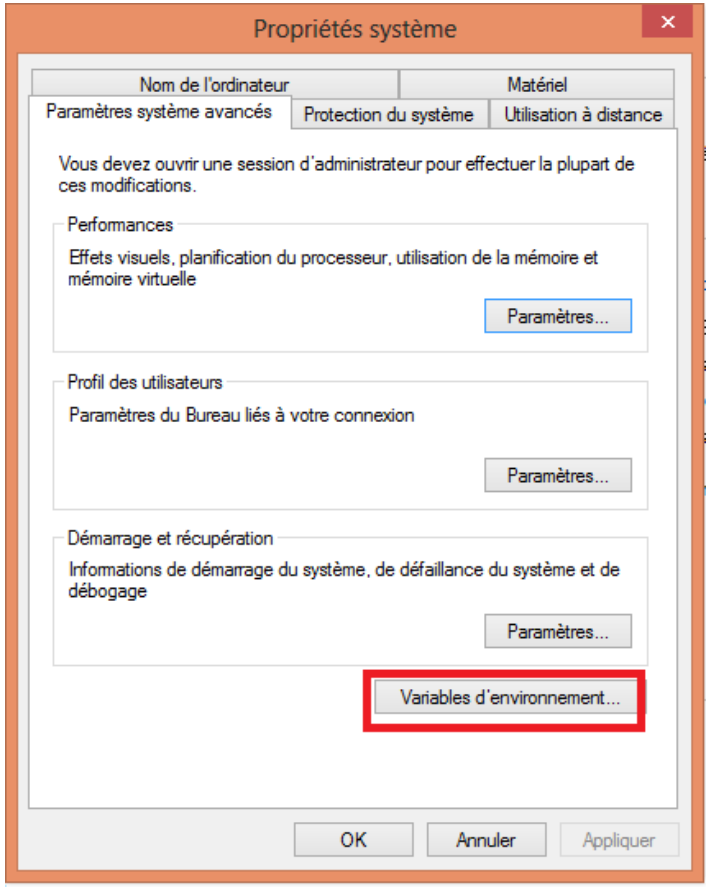
PARTIE 2

Exécuter un script depuis Python

2. Cliquer sur Paramètres système avancés.



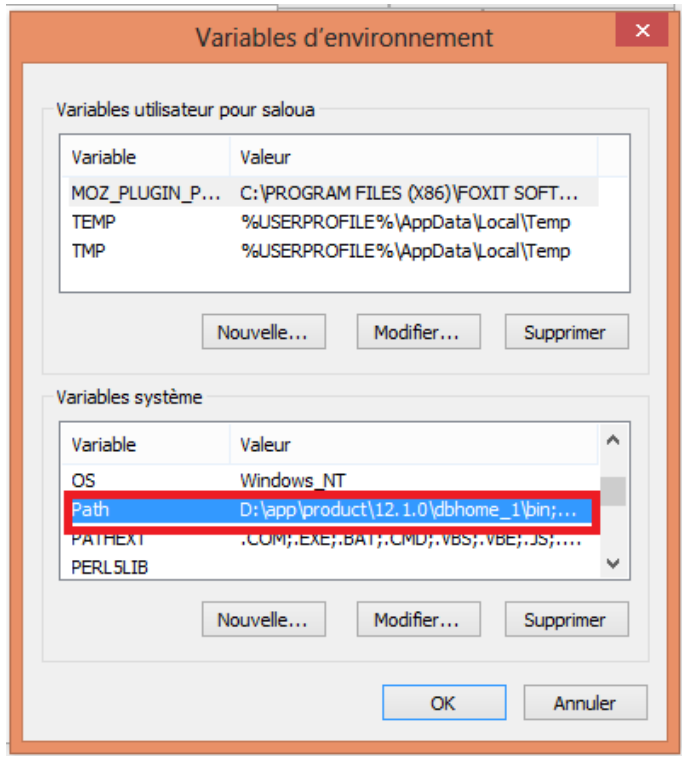
3. Cliquer sur Variables d'environnement.



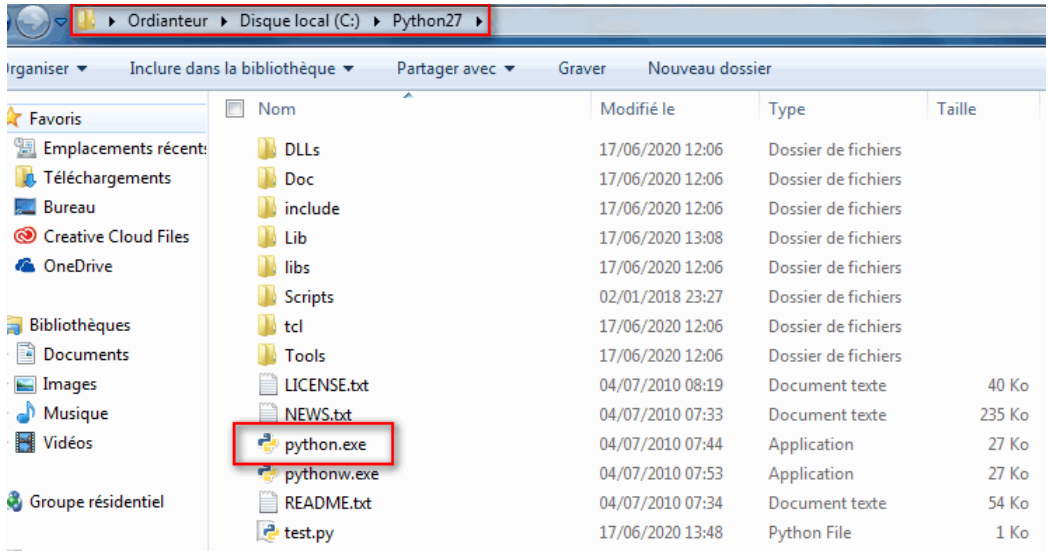
PARTIE 2

Exécuter un script depuis Python

4. Sous Variables système, choisir la variable Path puis cliquer sur Modifier.



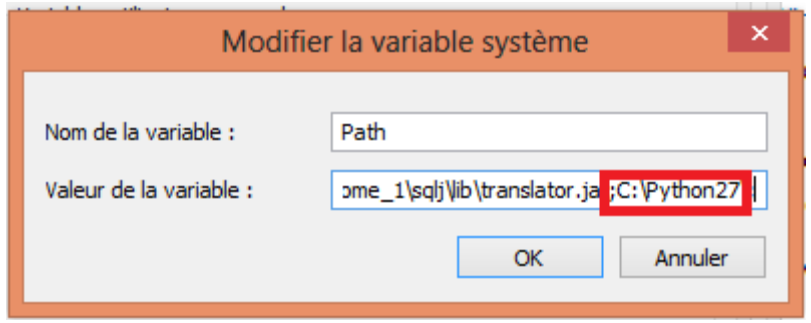
5. Copier le chemin du dossier C:\Python27 où se trouve le fichier exécutable de Python.



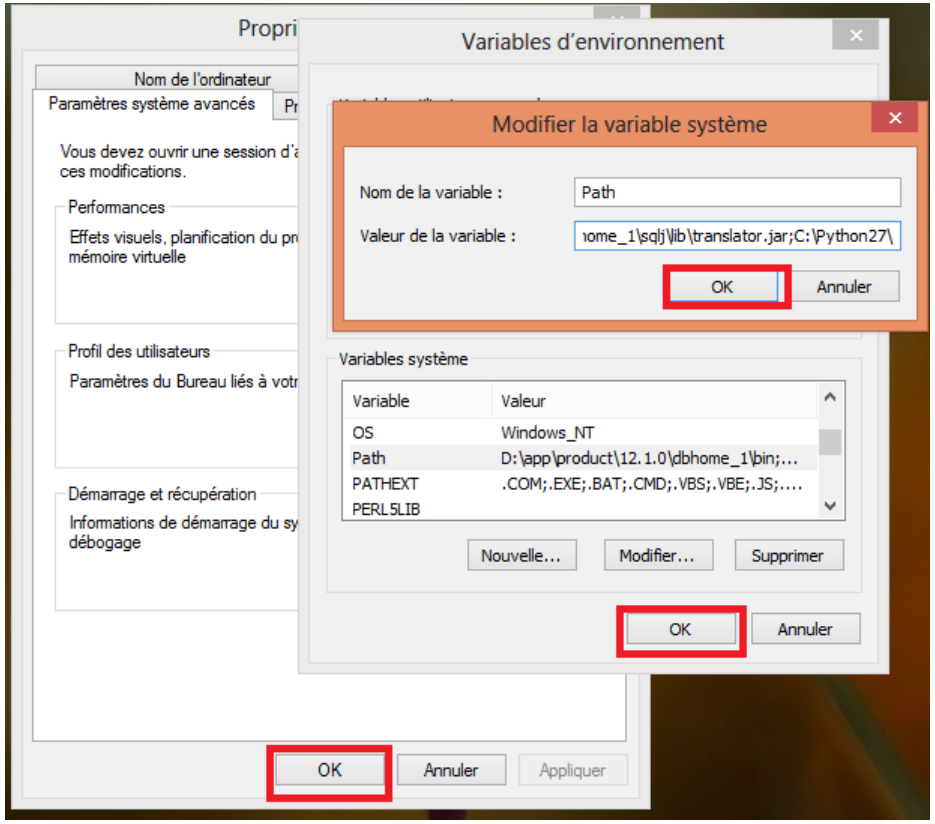
PARTIE 2

Exécuter un script depuis Python

6. Cliquer sur Modifier puis ajouter la valeur ;C:\Python27.



6. Cliquer sur le bouton OK.
7. Exécuter votre fichier Python en ligne de commande sous Windows.



PARTIE 2

Bilan

- Le développement de programmes nécessite :
  - L'installation d'un interpréteur, par exemple Python dans la version souhaitée.
  - L'installation d'un environnement de développement, par exemple Visual Studio Code.
- Plusieurs méthodes sont possibles :
  - Télécharger et installer les outils individuellement.
  - Utiliser un environnement complet comme Anaconda.
  - Utiliser un outil en ligne de commande.
- L'écriture d'un programme nécessite :
  - De créer un dossier pour stocker le code.
  - De créer un fichier avec l'extension .py
  - D'écrire un code valide et de l'enregistrer.
- L'exécution du programme s'effectue facilement depuis l'IDE Visual Studio Code en cliquant sur le bouton d'exécution.
  - On peut également retrouver les exécutions précédentes avec la flèche du haut du clavier directement dans le terminal.

À suivre

- Le chapitre suivant présente les éléments nécessaires à la création d'un script pour faciliter les opérations de gestion.

CHAPITRE 2

Créer un script pour faciliter les opérations de gestion

Ce que vous allez apprendre dans ce chapitre :

- Décrire les opérations de gestion
- Découvrir les structures relatives aux tâches de gestion (les modules spécifiques, listes, dictionnaires, …)
- Exécuter des scripts de gestion

04 heures

CHAPITRE 2

Créer un script pour faciliter les opérations de gestion

1. **Description des opérations de gestion**
2. Structures relatives aux tâches de gestion (les modules spécifiques, listes, dictionnaires, …)
3. Exécution de scripts de gestion
4. Ecriture et exécution en ligne de commandes

Opérations de gestion vs administration

- Les opérations d'administration d'un système consistent à assurer son maintien opérationnel : gestion des utilisateurs, sécurisation, configuration du réseau et du périphérique.
    - Nous abordons ces opérations dans la partie 3.
- Les opérations de gestion sont des opérations permettant de gérer l'exploitation *métier*. Par exemple :
    1. La gestion des logs d'un service installé sur la machine ;
    2. La suppression automatique des documents anciens pour respecter des règles de conformité ;
    3. L'archivage des données utilisateurs ;
    4. La recherche et le renommage de fichiers.

Des objets aux structures de données complexes

- Les tâches de gestion nécessitent des structures de données avancées, notamment pour regrouper plusieurs valeurs qui fonctionnent ensemble.
    - Nous avons déjà vu la notion d'objet, qui regroupe les attributs qui définissent ses caractéristiques.
    - Par exemple, un programme de gestion des fichiers peut définir une classe métier **Fichier** pour stocker les caractéristiques utiles des fichiers.
- Lorsqu'un programme traite **plusieurs données de même type**, il est utile de les regrouper dans une structure appropriée.

À suivre

- La suite de ce chapitre présente les structures suivantes :
    1. Les ensembles
    2. Les listes
    3. Les dictionnaires

CHAPITRE 2

Créer un script pour faciliter les opérations de gestion

1. Description des opérations de gestion
2. **Structures relatives aux tâches de gestion (les modules spécifiques, listes, dictionnaires, …)**
3. Exécution de scripts de gestion
4. Ecriture et exécution en ligne de commandes

Les dictionnaires et les ensembles en PYTHON

1. Les ensembles
2. Les dictionnaires : tableau associatif

Ensembles

- Un ensemble est une collection d'objets distincts et de types **hétérogènes**.
- Un ensemble est une structure **non ordonnée** : il n'est pas possible d'utiliser un indice (rang) ni de sélectionner un sous-ensemble.
- Un ensemble en PYTHON est un objet **mutable**.

Comment initialiser/créer un ensemble ?

- **Initialisation :** Il n'existe pas de syntaxe particulière pour les ensembles comme il en existe, par exemple, pour les listes et les tuples (crochets vides ou parenthèses vides). Les accolades vides sont relatives aux dictionnaires.
- La seule méthode possible consiste à utiliser la méthode set() pour l'initialisation.
- La méthode set(iterable) permet de créer un ensemble avec comme éléments, les éléments de l'itérable.

Comment initialiser/créer un ensemble ?

```
In [1]: E=set(); E# ensemble vide
Out [1]: set()
In [2]: type(E);
Out [2]: <class 'set'>
In [3]: E={}; type(E)# Attention, ceci n'est pas un ensemble vide
Out [3]: <class 'dict'>
In [4]: E={4,7,8};E
Out [4]: {4,7,8}
In [5]: E={'Bonjour', 56, 56};E # éléments distincts
Out [5]: {'Bonjour', 56}
In [6]: E=set('Bonjour'); E # L'itérable est une chaine
Out [6]: {'B','j','o','n''r','u'}
In [7]: E=set([1,4,8,8]);E # L'itérable est une liste
Out [7]: {1,4,8}
In [8]: E=set((4,7,8,8));E # L'itérable est un tuple
Out [8]: {4,7,8}
```

```
In [9]: E=set(range(11));E # L'itérable est un range
Out [9]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
In [10]: E=set((i,i) for i in range(11) if i%2==0);E # par compréhension
Out [10]: {(0,0), (6,6), (4,4), (10,10), (8,8), (2,2)}
```

Opérations sur les ensembles

```
In [1]: E=set(range(11))
In [2]: len(E);
Out [2]: 11
In [3]: 5 in E
In [3]: True
In [4]: 6 not in E
Out [4]: False
```

```
In [5]: for i in E:
print(i)
0
1
2
3
4
5
6
7
8
9
10
```

PARTIE 2

Opérations sur les ensembles

Intersection - Union

In [1]: {1,2,3} | {4,5,6} # le symoble | pour l'union

Out[1]: {1,2,3,4,5,6}

In [2]: {1,2,3}.union({4,5,6})# la méthode union

Out [2]: {1,2,3,4,5,6}

In [3]: {4,5} & {4,5,6} # le symoble & pour l'intersection

Out[3]: {4,5}

In [4]: {4,5}.intersection({4,5,6}) # la méthode intersection

Out [4]: {4,5}

Différence - Différence symétrique

In [1]: {4, 5, 6, 7, 8} - {4, 5, 6, 10} # le symoble - pour la différence

Out[1]: {8, 7}

In [2]: {4,5,6,7,8}.dierence({4, 5, 6, 10})# la méthode différence

Out [2]: {8, 7}

In [3]: {4,5,6,7,8} ^ {4, 5, 6, 10}# le symbole ^ pour la différence symétrique

Out [3]: {7,8,10}

In [4]: {4,5,6,7,8}.symmetric_difference({4, 5, 6, 10})# la méthode symmetric_difference

Out [4]: {7,8,10}

Opérations sur les ensembles

In [1]: {4, 5}.issubset({4, 5, 6, 10}) # teste si un ensemble est un sous-ensemble d'un autre ensemble

Out[1]: True

In [2]: {4,5,6,10}.issuperset({4, 5}) # teste si un ensemble est un super-ensemble d'un autre ensemble

Out [2]: True

In [3]: {4,5,6}.isdisjpoint({4, 5}) # teste si deux ensembles sont disjoints

Out [3]: False

Ajout - suppression

In [1]: E={1, 2, 3}

In [2]: E.add(4);E

Out [2]: {1,2,3,4}

In [3]: E.remove(3);E

Out [3]: {1,2,4}

Les dictionnaires

- Les dictionnaires sont le seul type associatif en PYTHON. Ces objets associatifs permettent de réaliser une correspondance entre valeurs de hashage (clés) et objets associés (valeurs). On parle de couples **<clef, valeur>.**
- Le dictionnaire n'est pas une structure ordonnée.

→ Pas de position (rang) d'une valeur dans le dictionnaire. Par contre, l'accès à une valeur se fait à travers la clef.

- Un dictionnaire est un objet mutable.

Comment initialiser/créer un dictionnaire ?

```
In [1]: D=dict(); D # dictionnaire vide
In [1]: {}
In [2]: type(D)
Out [2]: <class 'dict'>
In [3]: D={}; type(D)# dictionnaire vide
In [3]: <class 'dict'>
In [4]: D={1:'A',2:'B',3:'C',4:'D',5:'E'};D # ensemble de clef:valeur
Out [4]: {1:'A',2:'B',3:'C',4:'D',5:'E'}
In [5]: D=dict((i,i) for i in range(1,6)) # par compréhension
Out [5]: {1:1,2:2,2:2,4:4,5:5}
```

PARTIE 2

PARTIE 2

Opérations sur les dictionnaires

```
In [1]: D={1:'A',2:'B',3:'C',4:'D',5:'E'};D
Out [1]: {1:'A',2:'B',3:'C',4:'D',5:'E'}

In [2]: len(D)
In [2]: 5

In [3]: 5 in D #Attention la recherche se fait par clef
Out [3]: True

In [4]: 'A' in D
Out [4]: False

In [5]: D.get(1) # ou bien D[1], attention 1 est la clef et non pas la position
Out [5]: 'A'
```

Les méthodes keys(), values() et items()

```
In [1]: D.keys()
Out [1]: dict_keys([1,2,3,4,5])
In [2]: D.values()
In [2]: dict_values(['A','B','C','D','E'])
In [3]: D.items()
Out [3]: dict_items([(1,'A'),(2,'B'),(3,'C'),(4,'D'),(5,'E')])
```

```
for i in D.keys():
print(i)
1
2
3
4
5
```

Les méthodes keys(), values() et items()

```
for i in D.values():
    print(i)
A
B
C
D
E
```

```
for i in D.items():
    print(i)
(1,'A')
(2,'B')
(3,'C')
(4,'D')
(5,'E')
```

Les méthodes keys(), values() et items()

```
for i,j in D.items():
    print(i,j)
1 A
2 B
3 C
4 D
5 E
```

Les séquences

- Une **séquence** est une structure de données linéaire constituée d'une suite d'objets (éléments).
- Les séquences partagent toutes le même modèle d'accès (via des **index**) : un élément est accessible par son index. Un index désigne la position de l'élément dans la séquence.
- Les séquences en PYTHON :
    - **Les chaines de caractères** : suite d'objets de type homogène (caractères), délimitée par de simples ou doubles guillemets.
    - **Les listes :** suite d'objets de type hétérogène, délimitée par des crochets.
    - **Les tuples :** suite d'objets de type hétérogène, délimitée par des parenthèses.

- Les listes sont des objets **mutables** (modifiables), tandis que les chaines de caractères et les tuples sont des objets **non mutables** (non modifiables).

Comment initialiser/créer une séquence ?

Séquence vide

```
In [1]: L=list();l # ou bien L=[]
Out [1]: []

In [2]: C=str();C # ou bien C=”
Out [2]: ”

In [3]: T=tuple();T # ou bien
T=()
Out [3]: ()
```

Type d’une séquence

```
In [4]: type(L)
Out [4]: <class ’list’>

In [5]: type(C)
Out [5]: <class ’str’>

In [6]: type(T)
Out [6]: <class ’tuple’>
```

Type d'une séquence

```
L=[1,2,3,'a','B', 'Bonjour',[5,6,'ici']]
# une liste de 7 éléments

C='Bonjour les futurs ingénieurs'
# une chaine de caractères

T=(L,c,167,('A4',0))
# un tuple de 4 éléments
```

Taille - Accès à un élément de la séquence

- Une séquence peut contenir une autre séquence.
- On calcule la taille d'une séquence S (le nombre d'éléments) à l'aide de la fonction **len (len(S))**.
- On accède à chaque élément de la séquence par son index :
  - **S[0]** : premier élément de S
  - **S[len(S) - 1]** : dernier élément de S
- Lorsque l'index est négatif, le parcours est effectué à rebours à partir de la fin :
  - **S[-1]** : premier élément de S (de droite vers la gauche).
  - **S[- len(S)]** : dernier élément de S (de droite vers la gauche).

Exemple :

In [1]: len(L)

Out [1]: 7

In [2]: L[0] # ou L[-len(l)]

Out [2]: 1

In [3]: L[len(L)-1] # ou L[-1]

Out [3]: [5,6,'ici']

In [4]: len(C)

Out [4]: 29

In [5]: C[0] # ou C[-len(C)]

Out [5]: 'B'

Exemple : taille - Accès à un élément de la séquence

```
In [6]: C[len(C)-1] # ou C[-1]
Out [6]: 's'

In [7]: len(T)
Out [7]: 4

In [8]: T[0] # ou T[-len(T)]
Out [8]:
[1,2,3,'a','B','Bonjour',[5,6,'ici']]

In [9]: T[len(T)-1] # ou T[-1]
Out [9]: ('A4',0)

In [10]: L[-1][0]
Out [10]: 5

In [11]: T[0][-1]
Out [11]: [5,6,'ici']

In [12]: T[0][-1][1]
Out [12]: 6
```

Découpage ou slicing

- PYTHON permet le découpage en tranches avec la syntaxe suivante **S[debut : fin]**.
- **S[debut : fin]** désigne la sous-séquence de S constituée des éléments dont les index sont compris entre debut et fin -1.
- Quand **debut** n'est pas spécifié, alors la tranche commencera du début de S.
- Quand **fin** n'est pas spécifié, alors la tranche finira à la fin de S.
- La syntaxe **S[debut : fin : pas]** possède un troisième paramètre (par défaut, égal à 1), indiquant le pas de sélection.

Exemple

In [1]: L[:]

Out [1]:[1,2,3,'a','B','Bonjour',[5,6,'ici']

In [2]: L[3:]

Out [2]: ['a','B','Bonjour',[5,6,'ici']

In [3]: L[:5]

Out [3]: [1,2,3,'a','B']

In [4]: L[2:5]

Out [4]: [3,'a','B']

In [5]: L[::2]

Out [5]: [1,3,'B',[5,6,'ici']

Exemple : découpage ou slicing

```
In [6]: L[:5]
Out [6]: [1,2,3,'a','B']

In [7]: L[-5:]
Out [7]: [3,'a','B','Bonjour',[5,6,'ici']]
In [8]: L[:-5]
Out [8]:[1,2]

In [9]: L[-4:-1]
Out [9]: ['a','B','Bonjour']

In [10]: L[-6::2]
Out [10]: [2,'a','Bonjour']

In [11]: L[::-1]
Out [11]: [[5,6,'ici'],'Bonjour','B','a',3,2,1]
```

Opération de concaténation

- **L'opération de concaténation se note +**
- Syntaxe : S1 + S2
- Cette opération permet de concaténer deux séquences de même type.
- Le résultat est une nouvelle séquence qui contient les contenus combinés de S1 et S2.

Exemple :

```
In[12]: [1,2,3]+[4,5,6]
Out[12]: [1,2,3,4,5,6]
```

Opération de répétition

- **L'opération de répétition se note ***
- Syntaxe : S * n (*n* est un entier).
- Cette opération permet d'avoir une nouvelle séquence composée de *n* copies consécutives de S.

Exemple :

```
In[13]: 'Bonjour'*3
Out[13]: 'BonjourBonjourBonjour'
```

Appartenance (in, not in)

- On peut vérifier si un objet est membre ou non d'une séquence en utilisant les opérateurs **in/not in**.

Exemple :

In[14]: 'B' in c

Out[14]: True

In[15]: 'i' in L[6][2]

Out[15]: True

In[16]: 6 in L[6][1]

Out[16]: TypeError: argument of type 'int' is not iterable

Les méthodes count/index

- La méthode **L:count(obj)** retourne le nombre d'occurrence de **obj** dans S.
- La méthode **L:index(obj)** retourne la position de la première occurrence de obj dans S. Si obj n'existe pas, alors cette méthode génère une exception de type **ValueError**.

Exemple :

```
In[17]: L.count(6)
Out[17]: 0

In[18]: L.index(2)
Out[18]: 1

In[19]: L.index(6)
Out[19]: ValueError: 6 is not in list
```

Les séquences : récapitulatif

Opération	Fonction
len(S)	Nombre d'éléments de la séquence S
S[i]	Élément à la position i
S[len(S)-1]	Dernier élément de S
S[:]	Toute la séquence S
S[a:]	Éléments d'indice a jusqu'à la fin
S[:b]	Éléments d'indice 0 jusqu'à b - 1
S[a:b]	Éléments d'indice a jusqu'à b - 1
S[a:b:c]	Éléments d'indice a jusqu'à b (pas=c)

Opération	Fonction
S[:n]	Les n premiers éléments de S
S[-n:]	Les n derniers éléments de S
S[::-1]	La séquence S inversée
S * n	S répétée *n* fois
S1 + S2	Concatène les séquences S1 et S2
obj in S	Teste si obj est membre de S
obj not in S	Teste si obj n'est pas membre de S
S.count(obj)	Retourne le nombre d'occurrences de obj dans S
S.index(obj)	Retourne la position de obj dans S. Si obj n'est pas membre de S, cette méthode génère une erreur.

Liste

- Une liste en PYTHON est une structure de données mutable, c'est-à-dire qu'on peut modifier son contenu sans changer son adresse mémoire.
- Modifier une liste : ajouter/remplacer/supprimer un élément ou une suite d'éléments.
- Ajouter un élément : les méthodes append et insert permettent d'ajouter un élément dans une liste.
- Ces méthodes modifient la liste sur laquelle s'applique la méthode.
  - **L:append(obj)** ajoute obj à la fin de L.
  - **L:insert(i ; obj)** ajoute obj à la position i dans L.

Exemple :

```
In [1]: L=[1,2,3]
In [2]: L.append('cc')
In [3]: L
Out [3]: [1,2,3,'cc']
In [4]: L.insert(1,('bb',5))
In [5]: L
Out [5]: [1,('bb',5),2,3,'cc']
```

Extension d'une liste

- La méthode **extend** permet d'étendre le contenu d'une liste.
- **L:extend(seq)** ajoute le contenu de **seq (itérable)** à la fin de L.

Exemple

```
In [1]: L=[1,2,3]

In [2]: L.extend('Bonjour')

In [3]: L
Out [3]: [1,2,3,'B','o','n','j','o','u','r']

In [4]:
L.extend(('coucou',4))

In [5]: L
Out [5]: [1,2,3,'B','o','n','j','o','u','r','coucou',4]

In[6]: L.extend(67)
Out[6]: TypeError: 'int' object is not iterable
```

Suppression

- La méthode L:remove(obj) supprime la première occurrence de obj.
- La méthode L:pop() supprime le dernier élément dans L et le retourne.
- La méthode L:pop(i) supprime l'élément à la position i et le retourne.

Exemple :

In [1]: L=[1,2,3,4,5,6]

In [2]: L.remove(1)

In [3]: L

Out[3]: [2,3,4,5,6]

In [4]: L.pop()

Out [4]: 6

In [5]: L

Out [5]: [2,3,4,5]

In [6]: L.pop(1)

Out[6]: 3

In [7]: L

Out [7]: [2,4,5]

Suppression d'un élément

Exemple :

```
In [1]: L=[1,2,3,4,5,6]

In [2]: del L[4]# equivalent à L.pop(4)

In [3]: L
Out [3]: [1,2,3,4,6]
```

Suppression de plusieurs éléments

Exemple :

```
In [4]: del L[2:4]

In [5]: L
Out [5]: [1,2,6]
```

Modification d'un élément

Exemple :

In [1]: L=[1,2,3,4,'a','b','c','d']

In [2]: L[4]='A'

In [3]: L

Out [3]: [1,2,3,4,'A','b','c','d']

Modification de plusieurs éléments

Exemple :

In [4]: L[2:4]=30,40

In [5]: L

Out [5]: [1,2,30,40,'A','b','c','d']

Autres méthodes

- **L.sort()** trie les éléments de L.
- **L.reverse()** inverse les éléments de L.

Exemple

```
In [1]: L=[1,8,3,4,9,5,2]

In [2]: L.sort()

In [3]: L
Out [3]: [1,2,3,4,5,8,9]

In [4]: L.reverse()

In [5]: L
Out [5]: [9,8,5,4,3,2,1]
```

Création de liste - Par énumération

- Par énumération en utilisant la méthode ***list(iterable)***.

Exemple :

```
In [1]: L=list(range(0,11))

In [2]: L
Out[2] [0,1,2,3,4,5,6,7,8,9,10]

In [3]: L=list(range(0,11,2)

In[4]: L
Out [4]: [0,2,4,6,8,10]

In [5]: L=list('Bonjour')
In [5]: L
Out [5]: ['B','o','n','j','o','u','r']
```

Création de liste - Par compréhension

- La création de liste par compréhension permet de parcourir un objet itérable en renvoyant une liste.

Exemple :

```
In [1]: L=[i*2 for i in range(0,11)]

In [2]: L
Out[2]: [0,2,4,6,8,10,12,14,16,18,20]

In [3]: L=[i for i in 'Bonjour']

In[4]: L
Out [4]: ['B','o','n','j','o','u','r']

In [5]: L=[i+i for i in 'Bonjour']
In [6]: L
Out [6]: ['BB','oo','nn','jj','oo','uu','rr']
```

Création de liste - Par compréhension

- La création de liste par compréhension permet de renvoyer une liste dont le contenu peut-être filtré.

Exemple :

```
In [1]: L=[i for i in range(0,11) if i%2==0]

In [2]: L
Out[2] [0,2,4,6,8,10]

In [3]: L=[i for i in 'Bonjour' if i=='B' or i=='j']

In[4]: L
Out [4]: ['B','j']
```

Parcours de liste

Exemple :

```
L=[1, 'a', 'coucou',4]
for i in range(len(L)):
        print(L[i])
1
'a'
'coucou'
4

L=[1, 'a', 'coucou',4]
for i in L:
      print(i)
1
'a'
'coucou'
4
```

Copie d'un objet mutable

- Une affection x = y n'implique pas la création d'un nouvel objet, mais plutôt la création d'une nouvelle étiquette y (référence) vers l'objet déjà créé.
- Ceci ne pose pas de problème pour les objets non mutables. Par contre, ceci peut poser un problème pour les objets mutables tels que les listes, où la modification d'une liste implique la modification de l'autre liste.

Exemple :

```
In [1]: L1=[i for i in range(0,6)]
In [2]: L1
Out[2] [0,1,2,3,4,5]
In [3]: L2=L1
In[4]: L1[0]='A'
In [5]: L1
Out[5]: ['A',1,2,3,4,5]
In [6]: L2
Out [6]: ['A',1,2,3,4,5]
```

Solution : créer une copie de l'objet mutable indépendante de l'originale.

Différentes méthodes :

```
In [7]: L3=L1[:]
In [8]: L4=[i for i in L1]
In [9]: L5=list()
In[10]: for i in L1:
            L5.append(i)
In [11]: import copy as c
In[12]: L6=c.copy(L1)
```

➡ Vérifier les id des objets créés.

PARTIE 2

Copie superficielle/profonde

Si une liste contient des objets eux-mêmes mutables, la copie que nous venons d'effectuer est insuffisante.

➡ Copie superficielle

```
In [1]: L1=[[1,2,3],[4,5,6]]
In [2]: L2=L1[:]
In [3]: L1.append(56)
In[4]: L1,L2
Out [4]: [[1,2,3],[4,5,6],56],[[1,2,3],[4,5,6]]
In[5]: L1[0].append(66)
In[6]: L1,L2
Out[6]:[[1,2,3,66],[4,5,6],56],[[1,2,3,66],[4,5,6]]
```

Solution : effectuer une copie profonde, en utilisant la fonction **deepcopy** du module **copy**.

```
In [6]: L3=c.deepcopy(L1)
In [7]: id(L1[0])==id(L3[0])
Out [7]: False
In[8]: L1[0].append(80)
In[9]: L1,L3
Out[10]: [[1,2,3,66,80],[4,5,6],56],[[1,2,3,66],[4,5,6],56]
```

PARTIE 2

Les chaines - Rappel

- Les chaines de caractères : suite d'objets de type homogène (caractères), délimitée par de simples ou doubles guillemets.
- Les chaines sont des objets non mutables (non modifiables).
- Les chaines sont des séquences. Par conséquent, toutes les opérations relatives aux séquences sont applicables aux chaines :
    - L'accès à un élément/le découpage ;
    - La fonction len ;
    - L'appartenance d'un élément (in, not in) ;
    - Les opérations de concaténation/répétition ;
    - Les méthodes count/index.
- PYTHON offre plusieurs méthodes de manipulation de chaines de caractères.

Les chaines de caractères

Méthode	Description
c:split()	Retourne une liste composée des mots composant la chaine c. Par défaut, le séparateur de mots est le caractère espace.
c:split(':')	Même définition, sauf que cette fois, le séparateur de mots est le caractère ':'.
c:find(cc)	Détermine si cc est une sous-chaine de c. Retourne l'indice si cc est trouvé, -1 sinon.
c:find(cc,deb,fin)	Même définition, sauf que cette fois, la recherche commence à partir de la position *deb* et finit à la position fin -1.
c:replace(str 1,str2)	Remplace toutes les occurrences de str1 dans c par str 2.
c:replace(str 1,str 2,n)	Remplace les *n* premières occurrences de str1 dans c par str 2.

Les chaines de caractères

Méthode	Description
c.upper()	Convertit toutes les minuscules de c en majuscules.
c.lower()	Convertit toutes les majuscules de c en minuscules.
c.title()	Retourne une version de c où chaque initiale de chaque mot est une majuscule, comme dans un titre en anglais.
c.isupper()	Retourne True si c contient au moins un caractère alphabétique et que tous les caractères sont en majuscules, False, sinon.
c.islower()	Retourne True si c contient au moins un caractère alphabétique et que tous les caractères sont en minuscules, False, sinon.
c.istitle()	Retourne True si toutes les initiales des mots de c sont en majuscules, False, sinon.
c.isumeric()	Retourne True si c ne contient que des caractères numériques, False, sinon.
c.isalpha()	Retourne True si c contient au moins un caractère et que tous les caractères sont alphabétiques, False, sinon.
c.isalnum()	Retourne True si c contient au moins un caractère et que tous les caractères sont alphanumériques, False, sinon.

Remarques

- N'oubliez pas que la chaine est un objet non mutable. Par conséquent, toutes ces méthodes ne modifient pas la chaine c mais renvoient une nouvelle chaine.

Les tuples - Rappel

- Les **tuples** : suite d'objets de type hétérogène, délimitée par des parenthèses.
- Les tuples sont des objets **non mutables** (non modifiables).
- Les tuples sont des séquences. Par conséquent, toutes les opérations relatives aux séquences sont applicables aux tuples :
    - L'accès à un élément/le découpage ;
    - La fonction len ;
    - L'appartenance d'un élément (in, not in) ;
    - Les opérations de concaténation/répétition ;
    - Les méthodes count/index.

Exemple :

```
In [1]: T=tuple();T # ou bien T=() Tuple vide
Out [1]: ()
```

Les tuples - Rappel

```
In [2]: type(T)
Out [2]: <class 'tuple'>

In [3]: T=('Bonjour', 14, ['coucou', 15], (67, 'salut'));T # Tuple composé de 4 éléments
Out [3]: ('Bonjour', 14, ['coucou', 15], (67, 'salut'))

In [4]: T=('Bonjour');type(T) # Attention, ceci n'est pas un tuple
Out [4]: <class 'str'>

In [5]: T=('Bonjour',);type(T) # Tuple composé d'un seul élément, remarquez la virgule à la fin
Out [5]: <class 'tuple'>
```

PARTIE 2

Les tuples

- On peut également créer un tuple par énumération, en utilisant la méthode **tuple** avec, en paramètre, un objet itérable :

T=tuple(iterable)

- La création de tuple par compréhension est également possible.

Exemple :

```
In [6]: T=tuple('Bonjour');T
Out [6]: ('B', 'o', 'n', 'j', 'o', 'u', 'r')
In [7]: T=tuple(range(11));T
Out [7]: (0,1,2,3,4,5,6,7,8,9,10)
In [8]: T=tuple([1,6, ['Bonjour', 65]]);T
Out [8]: (1,6,['Bonjour',65])
In [9]: T=tuple((i,i) for i in range(11) if i%2==0);T
Out [9]: ((0,0), (2,2), (4,4), (6,6), (8,8), (10,10))
```

Conversion - Transtypage

- Les méthodes list(), str() et tuple() sont des méthodes de création respectivement de liste, de chaine et de tuple.
- Ces méthodes peuvent également être considérées comme des méthodes de transtypage, c'est-à-dire de conversion de types.

Exemple :

```
In [1]: T=(1, 'Bonjour', 80 )

In [2]: L=list(T);L
Out [2]: [1,'Bonjour',80]

In [3]: L=['Bonjour', 14, ['coucou', 15], (67, 'salut')]

In [4]: T=tuple(L);T
Out [4]: ('Bonjour', 14, ['coucou', 15], (67, 'salut'))
```

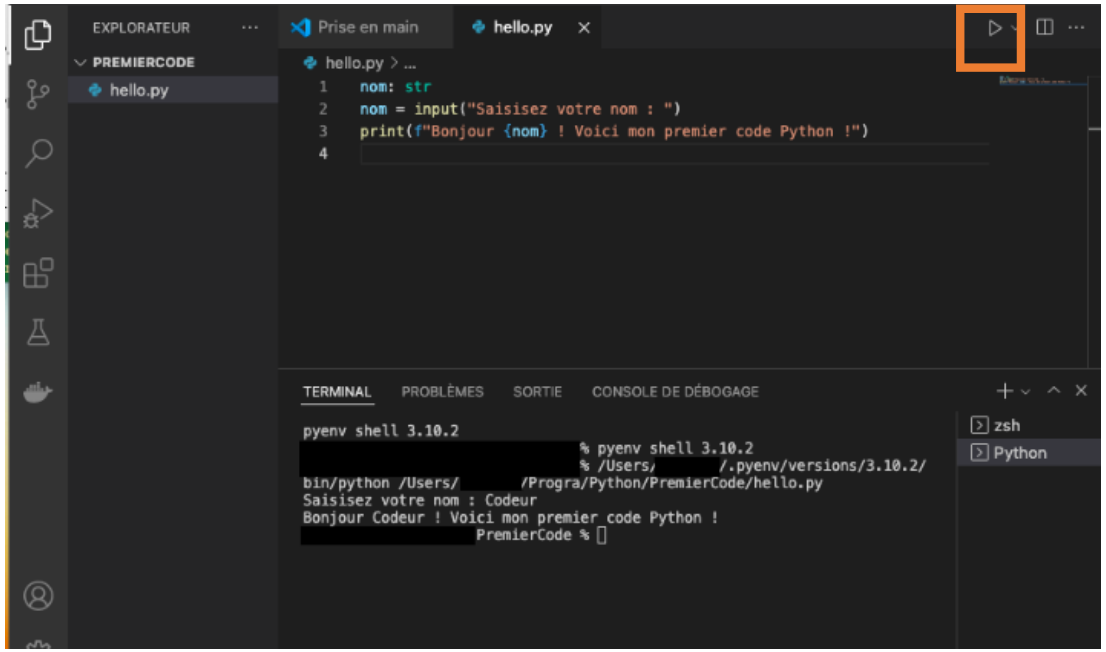
CHAPITRE 2

Créer un script pour faciliter les opérations de gestion

1. Description des opérations de gestion
2. Structures relatives aux tâches de gestion (les modules spécifiques, listes, dictionnaires, …)
3. **Exécution de scripts de gestion**
4. Ecriture et exécution en ligne de commandes

Rappel du chapitre précédent

- L'exécution du code depuis l'environnement de développement Visual Studio Code s'effectue de la manière suivante :
    1. Cliquez sur l'icone en forme de flèche de lecture (encadrée ci-contre).
    2. Le code s'exécute et s'arrête à la demande de saisie. Dans le terminal, saisissez un nom et validez.
    3. Le reste du code s'exécute.
    4. Pour relancer le code, vous pouvez répéter l'opération ou, dans le terminal, appuyer sur la flèche du haut pour retrouver la commande exécutée.



PARTIE 2

Exécution depuis le navigateur de fichiers

- Lorsqu'un script Python est présent dans le système de fichiers, il peut être exécuté de la manière suivante :
    1. Faire un clic-droit sur le fichier .py contenant le fichier ;
    2. Sélectionner **Ouvrir** *Avec* dans le menu et sélectionner p*ython.exe ;*
    3. Le script sera désormais exécuté par un double-clic, comme pour une application normale.

À suivre

- La dernière partie de ce chapitre montre comment lancer un script depuis la ligne de commande.

PARTIE 3

Appliquer l’administration système

Dans ce module, vous allez :

- Connaitre les commandes de base d’administration
- Administrer les ordinateurs à distance

09 heures

CHAPITRE 1

Connaitre les commandes de base d'administration

Ce que vous allez apprendre dans ce chapitre :

- Tests de commande d'administration Windows
- Exécution de commande d'administration Linux
- Ecriture de commande d'administration réseaux

05 heures

CHAPITRE 1

Connaitre les commandes de base d’administration

1. **Tests de commande d’administration Windows**
2. Exécution de commande d’administration Linux
3. Ecriture de commande d’administration réseaux

Rappels sur PowerShell

- PowerShell est un outil d'administration initialement conçu pour Windows, et désormais disponible en tant qu'outil multiplateforme
- PowerShell contient
    1. Un interpréteur de commandes, qui modernise l'ancienne « invite de commandes » Windows ;
    2. Un langage de script, qui permet d'automatiser des tâches en exécutant des commandes contenues dans un fichier ;
    3. Un outil de gestion de la configuration, dit « d'infrastructure as code ».
- PowerShell offre des commandes utiles pour l'administration d'un poste Windows
    - Le cmdlet *Get-Command* permet de lister les commandes disponibles.
    - La documentation officielle est située à : https://docs.microsoft.com/fr-fr/powershell/scripting/powershell-commands?view=powershell-7.2

Commandes utiles PowerShell pour l'administration

- PowerShell propose un module spécifique pour les commandes d'administration, Microsoft.PowerShell.Management, qui inclut notamment :
  - **Add-Computer :** ajoute l'ordinateur au domaine
  - **Add-Content :** ajoute du contenu à un fichier, par exemple de configuration
  - **Checkpoint-Computer :** crée un point de restauration sur l'ordinateur
  - **Get-EventLog :** liste les événements liés au système
  - **Get-Process :** liste les processus en cours sur le système
  - **Get-Service :** liste les services disponibles sur un ordinateur
  - **New-Item-Property :** ajoute une propriété dans un objet, comme un fichier ou la base de registres
  - **Remove-Item :** supprime un objet, comme un fichier ou une clé de la base de registres

- PowerShell propose également un module spécifique pour la gestion des utilisateurs, Microsoft.PowerShell.LocalAccounts, qui inclut par exemple :
  - **Enable-LocalUser :** active un compte utilisateur
  - **Set-LocalGroup :** modifie un groupe local de sécurité
  - **New-LocalUser :** crée un nouveau compte utilisateur

Commandes utiles PowerShell pour la manipulation de fichiers structurés

- Les fichiers structurés peuvent provenir de logs, ou être extraits de bases de données ou de tableurs.

Ils sont facilement manipulables.

- **Get-Content :** lit le contenu d'un fichier de manière brute. Pour lire et manipuler un fichier structuré comme un fichier CSV, des fonctions spécifiques sont disponibles. Par exemple **Import-Csv** lit et analyse le contenu d'un fichier CSV en tenant compte des délimiteurs.

Dans le code suivant,

```
PS> $firstCsvRow = Import-Csv -Path ./salaries.csv | Select-Object -First
PS> $firstCsvRow | Select-Object -ExpandProperty 'Prénom'
```

la première ligne du fichier est chargée et mise dans la variable, puis la valeur de l'attribut **Prénom** est récupérée.

- Lors de la lecture du fichier par **Import-Csv**, un *objet* de type **PSCustomObjects** est créé avec le contenu du fichier. Cet objet peut ensuite être manipulé.

Par exemple,

```
PS> Import-Csv -Path ./Salaries.csv | Where-Object {$_.Department -eq 'Production' }
```

affiche la liste des salariés du département Production

- Si le délimiteur du fichier CSV est une tabulation au lieu d'une virgule, il est possible de configurer la commande comme suit :

```
PS> Import-Csv -Path ./salaries.csv -Delimiter "`t »
```

PARTIE 3

Mise en œuvre des commandes PowerShell dans un script

- Un script PowerShell est un fichier qui contient la séquence de commandes à exécuter
- Les scripts permettent de mettre en œuvre les 4 briques algorithmiques :
    1. La gestion des variables ;
    2. La lecture/écriture ;
    3. Les conditions ;
    4. Les boucles.
- L'écriture d'un script peut s'effectuer dans Visual Studio Code, qui dispose d'une extension pour aider au développement, comme pour Python
- L'extension d'un script PowerShell est .PS1
- Le **Hello World !** d'un script PowerShell s'écrit avec la commande **Write-Host,** qui correspond à la fonction **print()** de Python

CHAPITRE 1

Connaitre les commandes de base d’administration

1. Tests de commande d’administration Windows
2. **Exécution de commande d’administration Linux**
3. Ecriture de commande d’administration réseaux

Terminal et ligne de commande

- Traditionnellement, l'administration Windows s'est effectuée avec une interface graphique. La puissance de PowerShell a été un facteur clé pour amener les administrateurs à prendre en mains la ligne de commande.
- À l'inverse, les administrateurs Linux ont pour la plupart débuté par la liste de commande. Les interfaces graphiques sont apparues en tant que sur-couche, pour rendre Linux accessible au grand public.
- Les distributions Linux proposent une palette d'émulateurs de terminaux qui peuvent être lancés depuis l'interface graphique. Il est également possible de se connecter directement en ligne de commande, sans lancer l'interface graphique. Cela est notamment utile en mode **recovery**, si le système est dans un état instable.
- Par défaut, l'interpréteur de commandes est bash. Comme PowerShell, bash permet
    - D'exécuter des commandes interactives l'une après l'autre ;
    - De créer un script pour exécuter des commandes séquentiellement. Le langage offre également les quatre briques algorithmiques.

Commandes de gestion utilisateur

- Différentes commandes permettent de gérer les utilisateurs
  - **adduser :** crée un nouveau compte utilisateur
  - **password –l :** désactive un compte utilisateur
  - **userdel :** supprime un compte utilisateur
  - **usermod :** ajoute un utilisateur à un groupe
  - **deluser :** supprime un utilisateur d'un groupe
  - **finger :** fournit les informations sur les utilisateurs connectés

Commandes de gestion des disques

- Différentes commandes permettent de gérer les disques
  - **df -h, -i :** fournit les statistiques du système de fichier
  - **mkfs -t -V :** crée un système de fichier
  - **resize2fs :** met à jour un système de fichier
  - **fsck -A –N :** vérifie un système de fichier
  - **pvcreate :** crée un volume physique
  - **mount -a –t :** monte un système de fichier
  - **fdisk –l :** édite une partition
  - **lvcreate :** crée un volume logique
  - **umount -f –v :** démonte un système de fichier

Commandes de manipulation de fichiers textes

- Différentes commandes permettent de manipuler des fichiers
  - **tr –d :** convertit ou supprime un caractère
  - **uniq :** exlcut les doublons dans un fichier
  - **split –l :** sépare les contenus d'un fichier
  - **wc –w :** compte le nombre de lignes, de mots et de caractères d'un fichier
  - **head –n :** affiche les n premières lignes d'un fichier
  - **cut –s :** supprime une section d'un fichier
  - **diff –q :** compare 2 fichiers, ligne par ligne
  - **join –i :** joint 2 fichiers
  - **less :** affiche le contenu d'un fichier
  - **tail :** affoche la dernière partie d'un fichier

CHAPITRE 1

Connaitre les commandes de base d’administration

1. Tests de commande d’administration Windows
2. Exécution de commande d’administration Linux
3. **Ecriture de commande d’administration réseaux**

Commandes d'administration réseau Windows

- **Ping :** teste la connexion avec une autre machine
- **Netstat :** fournit les informations sur les connexions réseau d'une machine
- **Arp :** gestion du cache ARP de la machine
- **Hostname :** fournit le nom de la machine sur le réseau
- **Tracert :** montre les chemin utilisé par un paquet IP entre 2 machines
- **IPConfig :** affiche la configuration IP de la machine
- **NSLookup :** fournit les informations sur le DNS
- **Route :** affiche la table de routage
- **NetDiag :** fournit un diagnostic réseau

Commandes d'administration réseau Linux

- **Ping :** teste la connexion avec une autre machine
- **Netstat :** fournit les informations sur les connexions réseau d'une machine
- **Ifconfig :** affiche et manipule les interfaces réseau
- **Traceroute :** indique le chemin d'un paquet IP entre 2 machines
- **dig :** fournit les informations sur le DNS
- **host :** effectue des requêtes DNS
- **curl :** télécharge un fichier
- **whois :** fournit les informations sur un nom de domaine
- **Ifplugstatus :** indique si un cable est connecté ou non

CHAPITRE 2

Administrer les ordinateurs à distance

Ce que vous allez apprendre dans ce chapitre :

- Connaissance aisée de connexion ssh et bureau à distance
- Tests de fonctionnement de commandes d'administration à distance

04 heures

CHAPITRE 2

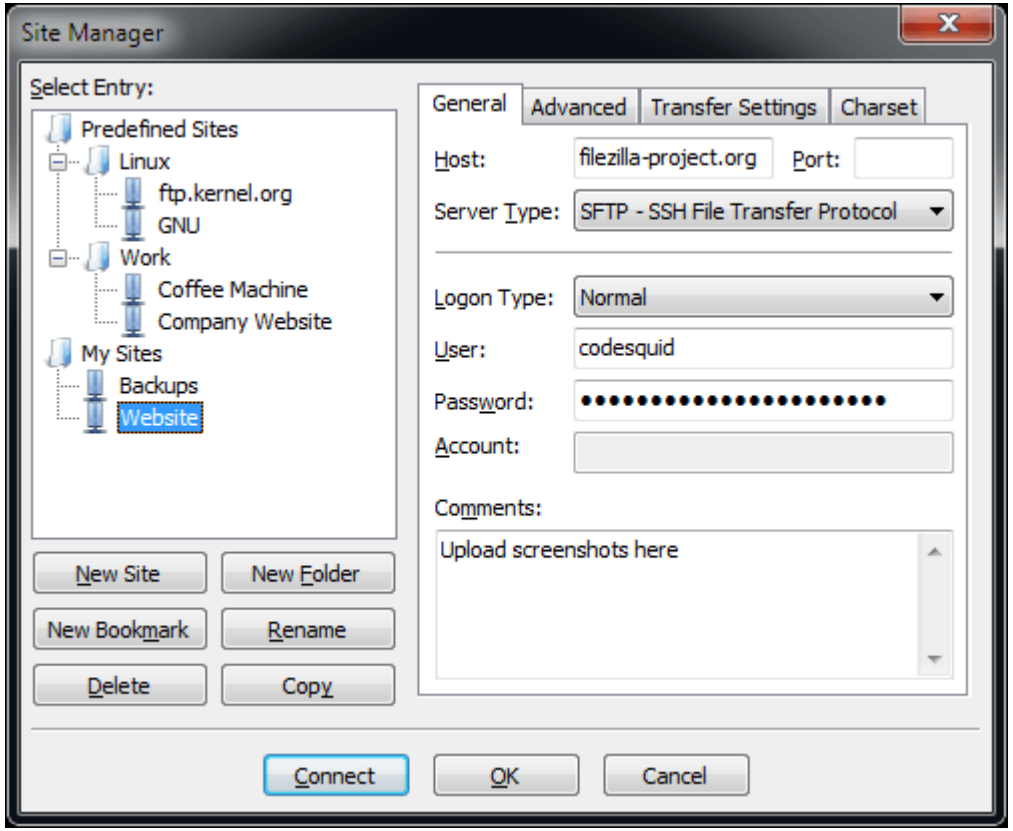
Administrer les ordinateurs à distance

1. **Connaissance aisée de connexion ssh et bureau à distance**
2. Tests de fonctionnement de commandes d'administration à distance

PARTIE 3

SSH

- SSH est un protocole sécurisé permettant d'établir une liaison cryptée entre 2 machines.
- L'outil le plus utilisé est OpenSSH, qui fournit l'utilitaire en ligne de commande *ssh.*
- Dans un terminal, ssh est utilisé de la manière suivante :
    - **ssh – v :** fournit des informations sur l'outil ;
    - **ssh –l login hote :** établit une connexion vers l'hôte **hote** avec le nom d'utilisateur *login* ;
    - **ssh –X :** permet d'établir une connexion avec l'interface graphique de l'hôte cible.
- OpenSSH fournit également des outils complémentaires, comme **scp** pour la copie sécurisée de fichiers et **sftp** pour le téléchargement sécurisé.
- Une connexion SSH peut également être établie à l'aide d'outils graphiques, comme FileZilla, illustré ci-contre.

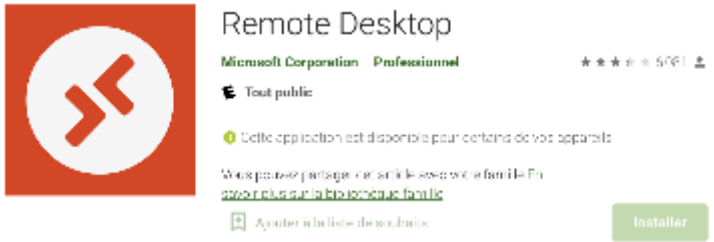


Bureau à distance : accorder l'accès

- L'accès à distance peut être activé nativement par les outils fournis dans Windows 10 Professionnel.
    - Pour vérifier l'édition de Windows, ouvrir les paramètres puis à propos> et rechercher Edition.
- Si la configuration est compatible,
    1. accéder aux paramètres puis **Système** > **Bureau** à distance et
    2. sélectionner Activer le Bureau **à distance.**
- Le nom du PC est indiqué sous **Procédure de connexion à cet ordinateur**.

Bureau à distance : se connecter

- La connexion à un ordinateur configuré pour accepter les connexions peut s'effectuer depuis un autre poste windows ou un équipement mobile sous Android/iOS.
- Depuis un autre poste windows,
    - Accéder à la zone de recherche de la barre des tâches ;
    - Saisir *Connexion au Bureau à Distance ;*
    - Sélectionner Connexion au Bureau à distance ;
    - Renseigner le nom du PC cible
- Depuis un équipement mobile Android/iOS, télécharger l'application bureau à distance sur le store approprié. Par exemple pour Android :



CHAPITRE 2

Administrer les ordinateurs à distance

1. Connaissance aisée de connexion ssh et bureau à distance
2. **Tests de fonctionnement de commandes d'administration à distance**

Test de connexion à distance avec PowerShell

- PowerShell fournit une cmdlet **Test-Connection** qui permet de tester la connexion avec une machine distante.
- L'exemple ci-dessous illustre l'utilisation de cette cmdlet :

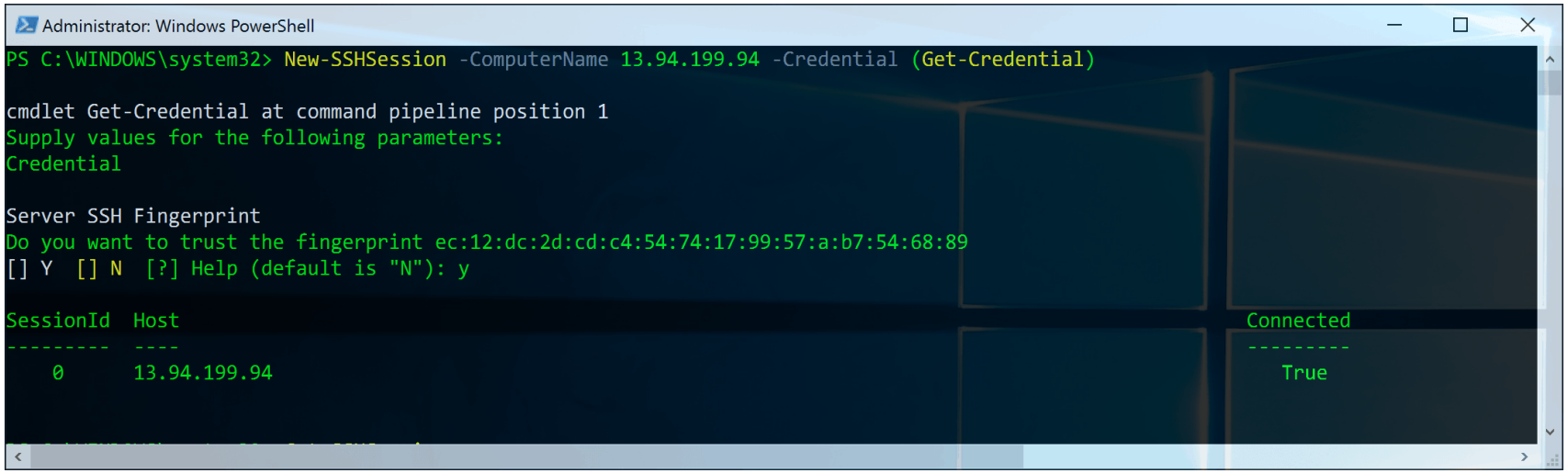
```
Test-Connection -TargetName Server01 -IPv4

Destination: Server01

Ping Source          Address                 Latency BufferSize Status
                                               (ms)        (B)
---- ------          -------                 ------- ---------- ------
   1 ADMIN1          10.59.137.44                24         32 Success
   2 ADMIN1          10.59.137.44                39         32 Success
   3 ADMIN1          *                            *          * TimedOut
   4 ADMIN1          10.59.137.44                28         32 Success
```

Connexion SSH depuis PowerShell

- Lorsque 2 machines peuvent communiquer, une connexion SSH peut être établie directement en PowerShell :



- Il est ensuite possible d'utiliser les commandes d'administration introduites au chapitre précédent.

Bilan

- Là où Python permet l'écriture de scripts de gestion *métiers,* des outils spécifiques permettent de gérer l'administration système
  - PowerShell pour Windows ;
  - Bash pour Linux.
- Il est également possible de gérer le réseau en ligne de commandes, via des commandes dédiées sous Windows ou Linux.
- SSH et le bureau à distance permettent d'administrer des machines distantes.
- PowerShell fournit des outils pour tester la connexion vers une machine distante puis établir une connexion sécurisée via le protocole SSH.

À suivre

- La section suivante traite de la création de programmes pour les tâches d'administration.

PARTIE 4

Créer des programmes pour les tâches d'administration

Dans ce module, vous allez :

- Automatiser les tâches redondantes
- Optimiser l'exécution des tâches d'administration

04 heures

CHAPITRE 1

Automatiser les tâches redondantes

Ce que vous allez apprendre dans ce chapitre :

- Spécification des tâches redondantes
- Création de scripts pour les tâches redondantes
- Tests de fonctionnement de scripts en conditions réelles
- Planification des tâches avec les outils système

02 heures

CHAPITRE 1

Automatiser les tâches redondantes

1. **Spécification des tâches redondantes**
2. Création de scripts pour les tâches redondantes
3. Tests de fonctionnement de scripts en conditions réelles
4. Planification des tâches avec les outils système

Pourquoi utiliser Python pour l'automatisation des tâches ?

- Python offre une grande lisibilité et une syntaxe accessible.
- Comparé à d'autres langages, Python se démarque clairement comme l'un des plus simples du groupe.

Exemple :

```
1 #include stdout
2
3 int main()
4 {
5    std::cout << "Hello!\n";
6 }
```

```
1 print("Hello!")
```

Le langage Python

LES AVANTAGES

- Les structures de données permettent de stocker et d'accéder aux données.
- Python en propose de nombreux types, notamment des listes, des dictionnaires, des tuples et des ensembles.
- Ces structures vous permettent de gérer les données facilement, efficacement et, lorsqu'elles sont choisies correctement, d'augmenter les performances du logiciel. De plus, les données sont stockées de manière sécurisée et cohérente.
- Automatiser presque tout avec Python : de l'envoi d'e-mails et du remplissage de PDF et CSV (si vous n'êtes pas familier avec ce format de fichier, je vous conseille de le vérifier, il est par exemple utilisé par Excel) à l'interaction avec des API externes et à l'envoi de requêtes HTTP. Quelle que soit votre idée, il est plus que probable que vous puissiez la réaliser en utilisant Python avec ses modules et ses outils.

Que pouvez-vous automatiser avec Python ?

- Presque tout ! Avec un peu de travail, pratiquement toute tâche répétitive peut être automatisée.
- Pour ce faire, vous n'avez besoin que de Python sur votre ordinateur et des bibliothèques pour un problème donné.

CHAPITRE 1

Automatiser les tâches redondantes

1. Spécification des tâches redondantes
2. **Création de scripts pour les tâches redondantes**
3. Tests de fonctionnement de scripts en conditions réelles
4. Planification des tâches avec les outils système

Exemple 1 : Lecture et écriture de fichiers

- La lecture et l'écriture de fichiers est une tâche que vous pouvez automatiser efficacement à l'aide de Python. Pour commencer, il vous suffit de connaitre l'emplacement des fichiers dans votre système de fichiers, leurs noms et le mode que vous devez utiliser pour les ouvrir.
- Utilisation de l'instruction with pour ouvrir un fichier : une approche très recommandée. Une fois le code with terminé, le fichier se ferme automatiquement et le nettoyage est fait.
- Chargement du fichier en utilisant la méthode **open(). open()** prend un chemin de fichier comme premier argument et un mode d'ouverture comme second. Le fichier est chargé en lecture seule (« r ») par défaut. Pour lire tout le contenu d'un fichier, utilisez la méthode **read()**.

```
In [1]: with open(“text_file.txt”) as f :

 ...: print(f.read())

 ...:

A simple text file.

With few lines.

And few words.
```

Exemple 1 : Lecture et écriture de fichiers

- Pour lire le contenu ligne par ligne, utiliser la méthode **readlines()** , elle enregistre le contenu dans une liste.

```
In [2]: with open(“text_file.txt”) as f:
 ...: print(f.readlines())
 ...:
[“A simple text file.\n”, “With few lines.\n”, “And few words.\n”]
```

- Vous pouvez également modifier le contenu d'un fichier. L'une des options pour le faire est de le charger en mode écriture ("w"). Le mode est sélectionné via le deuxième argument de la méthode **open()**. Mais, faites attention, car cela écrase le contenu original !

Exemple 1 : Lecture et écriture de fichiers

```
In [3]: with open(“text_file.txt”, “w”) as f:
    ...: f.write(“Some content”)
    ...:
In [4]: with open(“text_file.txt”) as f:
    ...: print(f.read())
    ...:
Some content
```

- Ouvrir le fichier en mode ajout (« a »), ce qui signifie que le nouveau contenu sera ajouté à la fin du fichier, laissant le contenu d'origine intact.

```
In [5]: with open(“text_file.txt”, “a”) as f:
    ...: f.write(“\nAnother line of content”)
    ...:
In [6]: with open(“text_file.txt”) as f:
    ...: print(f.read())
    ...:
Some content
Another line of content
```

Exemple 2 : Envoyer des e-mails

- Tâche qui peut être automatisée avec Python : l'envoi d'e-mails.
- Python est fourni avec la grande bibliothèque **smtplib**.
- smtplib : à utiliser pour envoyer des e-mails via le protocole **SMTP** (Simple Mail Transfer Protocol).
- Envoyer un e-mail à l'aide de la bibliothèque **smtplib** et du serveur SMTP de Gmail.
- On a besoin d'un compte de messagerie dans Gmail.
- Créer un compte Gmail séparé pour les besoins de ce script : vous devrez activer l'option Autoriser les applications moins sécurisées, ce qui permet aux autres d'accéder plus facilement à vos données privées.
- Configurer le compte.

Exemple 2 : Envoyer des e-mails

1. Établissement d'une connexion SMTP.

```
In [1]: import getpass
In [2]: import smtplib
In [3]: HOST = "smtp.gmail.com"
In [4]: PORT = 465
In [5]: username = "username@gmail.com"
In [6]: password = getpass.getpass("Provide Gmail password: ")
Provide Gmail password:
In [7]: server = smtplib.SMTP_SSL(HOST, PORT)
```

- Les modules intégrés requis sont importés au début du fichier, nous utilisons **getpass** pour demander en toute sécurité le mot de passe et **smtplib** pour établir une connexion et envoyer des e-mails. Dans les étapes suivantes, les variables sont définies. HOST et PORT sont tous deux requis par Gmail : ce sont les constantes.

Exemple 2 : Envoyer des e-mails

2. **Saisir le nom de compte Gmail qui sera stocké dans la variable nom d'utilisateur et le mot de passe.**

- Il est recommandé de saisir le mot de passe à l'aide du module **getpass**. Il invite l'utilisateur à saisir un mot de passe et ne le renvoie pas une fois que vous l'avez saisi.
- Le script démarre une connexion SMTP sécurisée à l'aide de la méthode **SMTP_SSL().** L'objet SMTP est stocké dans la variable serveur.

```
In [8]: server.login(username, password)
Out[8]: (235, b'2.7.0 Accepted')
In [9]: server.sendmail(
 ...: "from@domain.com",
 ...: "to@domain.com",
 ...: "An email from Python!",
 ...: )
Out[9]: {}
In [8]: server.quit()
Out[8]: (221, b'2.0.0 closing connection s1sm24313728ljc.3 - gsmtp')
```

3. **Authentification :** grâce à la méthode **login()**.

4. **Envoyer des e-mails** : utilisation de la méthode **sendmail()**.

5. **Quitter l'application :** utilisation de la méthode **quit()**.

PARTIE 4

CHAPITRE 1

Automatiser les tâches redondantes

1. Spécification des tâches redondantes
2. Création de scripts pour les tâches redondantes
3. **Tests de fonctionnement de scripts en conditions réelles**
4. Planification des tâches avec les outils système

Tester un script

- **Test automatisé vs Test manuel :**
  - Vous avez probablement déjà créé un test sans vous en rendre compte. Vous souvenez-vous quand vous avez exécuté votre application et que vous l'avez utilisée pour la première fois ? Avez-vous vérifié les fonctionnalités et expérimenté leur utilisation ?
  - C'est ce qu'on appelle les tests exploratoires et c'est une forme de test manuel. Les tests exploratoires sont une forme de test qui se fait sans plan. Dans un test exploratoire, vous explorez simplement l'application.
  - Pour disposer d'un ensemble complet de tests manuels, il vous suffit de dresser une liste de toutes les fonctionnalités de votre application, des différents types d'entrées qu'elle peut accepter et des résultats attendus. Désormais, chaque fois que vous modifiez votre code, vous devez parcourir chaque élément de cette liste et le vérifier.

➡ C'est là qu'interviennent les tests automatisés. Les tests automatisés sont l'exécution de votre plan de test (les parties de votre application que vous souhaitez tester, l'ordre dans lequel vous souhaitez les tester et les réponses attendues) par un script au lieu d'un humain . Python est déjà livré avec un ensemble d'outils et de bibliothèques pour vous aider à créer des tests automatisés pour votre application.

- **Test unitaire vs Test d'intégration :**
  - Les tests unitaires sont utilisés pour tester de petits composants à l'intérieur du script. Un test d'intégration, quant à lui, est utilisé pour tester les composants qui dépendent les uns des autres.
  - Considérons une analogie: nous avons une maison et l'ampoule a grillé. Nous pouvons tester si notre lumière est soufflée ou non en basculant notre interrupteur. Ceci serait considéré comme un test unitaire. Cependant, si nous devions tester le système électrique sur lequel repose cette lumière, cela serait considéré comme un test d'intégration. Les tests d'intégration testent plusieurs composants en même temps.
  - Vous pouvez écrire à la fois des tests d'intégration et des tests unitaires en Python.

Tester un script

Test unitaire vs Test d'intégration :

Exemple :

- Pour écrire un test unitaire pour la fonction sum(), vous devez vérifier la sortie de sum() par rapport à une sortie connue.
- Vérifier que la somme() des nombres (1, 2, 3) est égale à 6 :

Python >>>

```
>>> assert sum([1, 2, 3]) == 6, "Should be 6"
```

- Cela n'affichera rien sur le REPL car les valeurs sont correctes.
- Si le résultat de sum() est incorrect, cela échouera avec une AssertionError et le message "Should be 6". Essayez à nouveau une instruction d'assertion avec les mauvaises valeurs pour voir une AssertionError :

Python >>>

```
>>> assert sum([1, 1, 1]) == 6, "Should be 6"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Should be 6
```

Tester un script

- Au lieu de tester sur le REPL, le mettre dans un nouveau fichier Python appelé test_sum.py et l'exécuter à nouveau :

Python

```python
def test_sum():
    assert sum([1, 2, 3]) == 6, "Should be 6"

if __name__ == "__main__":
    test_sum()
    print("Everything passed")
```

- Vous avez maintenant écrit un scénario de test, une assertion et un point d'entrée (la ligne de commande). Vous pouvez maintenant exécuter ceci en ligne de commande :

Shell

```
$ python test_sum.py
Everything passed
```

- Vous pouvez voir le résultat réussi, tout est passé.

Tester un script

- En Python, sum() accepte n'importe quel itérable comme premier argument. Vous avez testé avec une liste. Maintenant, testez également avec un tuple. Créez un nouveau fichier appelé test_sum_2.py avec le code suivant :

Python

```python
def test_sum():
    assert sum([1, 2, 3]) == 6, "Should be 6"

def test_sum_tuple():
    assert sum((1, 2, 2)) == 6, "Should be 6"

if __name__ == "__main__":
    test_sum()
    test_sum_tuple()
    print("Everything passed")
```

Tester un script

- Lorsque vous exécutez test_sum_2.py, le script génère une erreur car la somme de (1, 2, 2) est 5 et non 6. Le résultat du script vous donne le message d'erreur, la ligne de code et le retraçage :

Shell

```
$ python test_sum_2.py
Traceback (most recent call last):
  File "test_sum_2.py", line 9, in <module>
    test_sum_tuple()
  File "test_sum_2.py", line 5, in test_sum_tuple
    assert sum((1, 2, 2)) == 6, "Should be 6"
AssertionError: Should be 6
```

- Vous pouvez voir comment une erreur dans votre code génère une erreur sur la console avec des informations sur l'emplacement de l'erreur et le résultat attendu.
- Écrire des tests de cette manière est acceptable pour une simple vérification, mais que se passe-t-il si plusieurs échouent ? C'est là qu'interviennent les lanceurs de tests. Le lanceur de tests (test runner) est une application spéciale conçue pour exécuter des tests, vérifier la sortie et vous donner des outils pour déboguer et diagnostiquer les tests et les applications.

Choisir un testeur

- Il existe de nombreux testeurs disponibles pour Python. Celui intégré à la bibliothèque standard Python s'appelle **unittest**.
- Les principes d'unittest sont facilement transférables à d'autres frameworks. Les trois testeurs les plus populaires sont :
    - Unittest
    - nose ou nose2
    - pytest
- Il est important de choisir le meilleur testeur pour vos besoins et votre niveau d'expérience.

CHAPITRE 1

Automatiser les tâches redondantes

1. Spécification des tâches redondantes
2. Création de scripts pour les tâches redondantes
3. Tests de fonctionnement de scripts en conditions réelles
4. **Planification des tâches avec les outils système**

Planification des tâches

- Pour **exécuter dans vos codes python des taches de manières répétitives, Python** peut vous économiser beaucoup de lignes de code et de temps.
- Python intègre plusieurs fonctionnalités
- Exemples:
    - si votre planificateur est redémarré, il exécutera toutes les tâches qu'il aurait dû exécuter pendant qu'il était hors ligne.
    - arrêt programmé ou, si vous travaillez avec des connexions réseau ou un pare-feu, vous pouvez créer une tâche unique pour annuler les modifications au cas où vous vous tromperiez et vous verrouilleriez hors du système.

Solution intégrée

- **Le module sched :**
  - est un module très simple ;
  - peut être utilisé pour planifier des tâches ponctuelles pendant un certain temps.

Exemple :

```
import sched
import threading
import time

scheduler = sched.scheduler(time.time, time.sleep)

def some_deferred_task(name):
    print('Event time:', time.time(), name)

print('Start:', time.time())

now = time.time()
#      delay in seconds -----v  v----- priority
event_1_id = scheduler.enter(2, 2, some_deferred_task, ('first',))
event_2_id = scheduler.enter(2, 1, some_deferred_task, ('second',))
event_3_id = scheduler.enter(5, 1, some_deferred_task, ('third',))

# Start a thread to run the events
t = threading.Thread(target=scheduler.run)
t.start()

# Event has to be canceled in main thread
scheduler.cancel(event_2_id)

# Terminate the thread when tasks finish
t.join()

# Output:
# Start: 1604045721.7161775
# Event time: 1604045723.718353 first
# Event time: 1604045726.7194896 third
```

Solution intégrée

- Le code de l'exemple précédant définit le planificateur, qui est utilisé pour créer des événements (.enter) à exécuter.
- Chaque événement (appel à .enter) reçoit 4 arguments :
    - délai en secondes (en combien de secondes l'événement se produira-t-il?) ;
    - priorité: cet argument est intéressant si 2 événements sont programmés pour se produire exactement au même moment, mais ils doivent être exécutés séquentiellement. Dans ce cas, l'événement avec la priorité la plus élevée (numéro le plus bas) passe en premier ;
    - nom de la fonction à appeler ;
    - arguments de fonction optionnels.
- Dans cet extrait de code, nous pouvons également voir que la méthode .enter renvoie l'ID d'événement. Ces ID peuvent être utilisés pour annuler des événements comme illustré avec scheduler.cancel(event_2_id).

La bibliothèque Crontab

- Il existe un certain nombre de bibliothèques pour exécuter des tâches récurrentes en utilisant Python.
    - La bibliothèque python-crontab peut être installée avec pip install python-crontab ;
    - python-crontab: contrairement aux autres bibliothèques, crée et gère de véritables crontabs sur les systèmes Unix et des tâches sous Windows ;
    - Par conséquent, il ne s'agit pas d'émuler le comportement de ces outils du système d'exploitation, mais plutôt de les exploiter et d'utiliser ce qui existe déjà ;
    - Exemple: La raison courante de l'exécution de tâches récurrentes peut être la vérification de l'état du serveur de base de données. Cela peut généralement être fait en se connectant et en se connectant à la base de données et en exécutant une requête comme SELECT 1, comme suit.

La bibliothèque Crontab

```
from crontab import CronTab

# user=True denotes the current user
cron = CronTab(user=True)
job = cron.new(command='PGPASSWORD=test psql -U someuser -d somedb -c "SELECT 1" -h localhost')
job.setall("*/5 * * * *")

if cron[0].is_valid():  # If syntax is valid, write to crontab
    cron.write()

# crontab -l  # Check real crontab from shell
# */5 * * * * PGPASSWORD=test psql -U someuser -d somedb -c "SELECT 1" -h localhost
```

- python-crontab inclut la syntaxe cron.
- Pour définir l'horaire, on utilise la .setall méthode pour définir tous les champs.
- Avant de définir le calendrier, nous devons créer le crontab en utilisant CronTab() et spécifier l'utilisateur propriétaire. Si True est passé, l'ID de l'utilisateur exécutant le programme sera utilisé. Nous devons également créer un job individuel (.new()) dans cette crontab en passant command pour être exécuté et éventuellement aussi un fichier comment.
- Lorsque nous avons le crontab et son travail prêts, nous devons l'écrire, mais c'est une bonne idée de vérifier sa syntaxe .is_valid() avant de le faire.

CHAPITRE 2

Optimiser l’exécution des tâches d’administration

Ce que vous allez apprendre dans ce chapitre :

- Identification des erreurs des manipulations par la méthode manuelle
- Avantages de l’automatisation des tâches
- Réduction de temps de réalisation de tâche par l’automatisation

02 heures

CHAPITRE 2

Optimiser l’exécution des tâches d’administration

1. **Identification des erreurs de manipulations par la méthode manuelle**
2. Avantages de l’automatisation des tâches
3. Réduction de temps de réalisation de tâche par l’automatisation

Qu'est-ce que le test manuel ?

- Les tests manuels, comme le terme l'indique, font référence à un processus de test dans lequel une assurance de qualité teste manuellement l'application logicielle afin d'identifier les bogues. Pour ce faire, les assurances de qualité suivent un plan de test écrit qui décrit un ensemble de scénarios de test uniques. L'assurance qualité est nécessaire pour analyser les performances de l'application Web ou mobile du point de vue de l'utilisateur final.
- Les assurances de qualité vérifient le comportement réel du logiciel par rapport au comportement attendu, et toute différence est signalée comme un bogue.
- Le test manuel comprend le test manuel d'un logiciel, c'est-à-dire sans utiliser d'outil automatisé ou de script. Dans ce type, le testeur assume le rôle d'un utilisateur final et teste le logiciel pour identifier tout comportement ou bogue inattendu. Les tests manuels comportent différentes étapes, telles que les tests unitaires, les tests d'intégration, les tests système et les tests d'acceptation par les utilisateurs.
- Les testeurs utilisent des plans de test, des cas de test ou des scénarios de test pour tester un logiciel afin de garantir l'exhaustivité des tests. Les tests manuels comprennent également des tests exploratoires, car les testeurs explorent le logiciel pour y identifier les erreurs.

Test d'automatisation Vs. Test manuel

- Les tests manuels sont effectués manuellement par un analyste QA (humain) tandis que les tests automatisés sont effectués à l'aide de scripts, de code et d'outils d'automatisation (ordinateur) par un testeur.
- Le processus de test manuel n'est pas précis en raison des possibilités d'erreurs humaines, tandis que le processus d'automatisation est fiable car il est basé sur du code et des scripts.
- Le test manuel est un processus qui prend du temps alors que le test automatisé est très rapide.
- Les tests manuels sont possibles sans connaissances en programmation, tandis que les tests automatisés ne sont pas possibles sans connaissances en programmation.
- Les tests manuels permettent des tests aléatoires, tandis que les tests automatisés ne permettent pas de tests aléatoires.

CHAPITRE 2

Optimiser l’exécution des tâches d’administration

1. Identification des erreurs de manipulations par la méthode manuelle
2. **Avantages de l’automatisation des tâches**
3. Réduction de temps de réalisation de tâche par l’automatisation

1. Réduire les erreurs manuelles

- La saisie manuelle des données augmente les erreurs humaines. Ainsi, le premier des avantages de l'automatisation est qu'elle peut prendre en charge des tâches de saisie manuelle de données en votre nom, ce qui signifie un risque réduit d'erreur humaine. Cela garantit une qualité constante dans votre saisie de données.

Exemple :

- **Beautiful Soup** est une bibliothèque Python qui peut être utilisée pour extraire des données de fichiers HTML et XML. Au lieu de parcourir manuellement une tonne de fichiers HTML et de rechercher manuellement les informations dont vous avez besoin, vous pouvez utiliser cette bibliothèque pour vous faire gagner du temps et du travail.

2. Réactions rapides

- Un autre des avantages de l'automatisation est que l'outil permet des réactions rapides lorsqu'elles sont le plus nécessaires. Un outil d'automatisation des tâches peut analyser les données entrantes: vos e-mails, vos formulaires Web, les médias sociaux, les données système, etc. Ensuite, il peut vous alerter de choses comme des plaintes ou des pannes de service. L'automatisation peut également répondre aux messages entrants pour vous.

Exemple :

- smtplib est une excellente bibliothèque Python qui vous aidera à automatiser vos e-mails. Il utilise le protocole de transfert de courrier simple, qui peut être facilement intégré à la plupart des principales plates-formes de messagerie, telles que Gmail.
- Selenium est un outil open source que vous pouvez utiliser pour automatiser les tests effectués dans les navigateurs Web. Cela peut tester les sites Web pour les bugs, les plantages de site. Cela peut potentiellement faire gagner beaucoup de temps aux utilisateurs ou vous éviter de ne pas réaliser que votre site Web est hors ligne.

CHAPITRE 2

Optimiser l’exécution des tâches d’administration

1. Identification des erreurs de manipulations par la méthode manuelle
2. Avantages de l’automatisation des tâches
3. **Réduction de temps de réalisation de tâche par l’automatisation**

Réduction de temps de réalisation

- Les fichiers batch (de traitement par lots) et les scripts sont utilisés pour automatiser les processus dans tous les types d'environnements informatiques. Par exemple, les scripts sont conçus pour automatiser les tâches de routine telles que la sauvegarde et l'effacement des journaux d'événements, les tâches quotidiennes de réseau, la surveillance des performances du système et la création de rapports, ainsi que les modifications du registre.
- L'automatisation des scripts peut également aider à gérer les comptes utilisateur, les comptes de postes de travail, les applications et les services. Les équipes informatiques emploient souvent des programmeurs hautement qualifiés pour écrire et maintenir ces scripts, qui automatisent des tâches qui autrement, seraient traitées manuellement. Pourtant, de nombreux professionnels de l'informatique en interne et administrateurs système en solo prennent en main l'automatisation des scripts.

PARTIE 5

Créer des fichiers logs

Dans ce module, vous allez :

- Comprendre la persistance des données
- Manipuler les fichiers logs
- Tester le fonctionnement des scripts

04 heures

CHAPITRE 1

Comprendre la persistance des données

Ce que vous allez apprendre dans ce chapitre :

- Flux vers les fichiers
- Méthodes de création, lecture, écriture, modification et suppression dans un fichier

02 heures

CHAPITRE 1

Comprendre la persistance des données

1. **Flux vers les fichiers**
2. Méthode de création, lecture, écriture, modification et suppression dans un fichier

Motivation

- Les données manipulées sont introduites par l'utilisateur au moment de l'exécution.
- Le résultat est perdu dès que le programme s'arrête.

→ Les fichiers sont un excellent moyen pour la sauvegarde des données.

Objectif :

- Programmes qui récupèrent les données à partir d'un fichier et qui sauvegardent le résultat dans un fichier.
- PYTHON dispose d'un objet de type **file** qui va permettre de sauvegarder et de récupérer les données à partir d'un fichier.
- Les opérations sur un fichier :
    - Ouverture/Création
    - Fermeture
    - Lecture
    - Écriture

Ouverture/Création

Syntaxe :

- f=open(filename; mode)
- filename : une chaine de caractères indiquant le nom du fichier à ouvrir (exemple : ‘exp.txt').
- mode : le mode d'ouverture du fichier est donné sous forme de chaine de caractères. Voici les principaux modes d'ouverture :
    - **'r' :** ouverture en lecture seule (read). Si le fichier n'existe pas, une erreur IOError est levée.
    - **'w' :** ouverture en écriture (write). Si le fichier existe, alors il sera écrasé, sinon il sera créé.
    - **'a' :** ouverture en écriture en mode ajout (append). Si le fichier existe, l'écriture sera faite à la fin du fichier.
    - Sinon, le fichier est créé.
- On peut parler de mode 'rb', 'wb' et 'ab' pour ouvrir le fichier en mode binaire. Nous verrons un peu plus loin ce mode particulier avec le module **pickle**.

CHAPITRE 1

Comprendre la persistance des données

1. Flux vers les fichiers
2. **Méthode de création, lecture, écriture, modification et suppression dans un fichier**

Ouverture/Création

- Pour ouvrir en lecture, le fichier ‘exp.txt' du répertoire courant : f=open(‘exp.txt','r')
- Si le fichier existe dans un autre emplacement, le nom du fichier doit être précédé par son chemin d'emplacement.

Exemple :

- f=open('/home/saloua/Bureau/exp.txt','r')

NB

- Le répertoire courant peut-être changé : voir Section module os.

Attention

- Ne pas confondre le nom physique du fichier 'test.txt' et le nom logique f utilisé par la suite dans le programme. f fait référence au fichier ‘exp.txt' : les caractères/lignes sont lus séquentiellement, c'est-à-dire les uns après les autres, sans possibilité de retour en arrière ni de saut en avant.

Fermeture

Syntaxe :

```
f.close()
```

NB

- N'oubliez pas de fermer un fichier après l'avoir lu !

PARTIE 5

Lecture

- Pour lire le contenu d'un fichier, on utilise l'une des méthodes suivantes :
  - **f.read() :** permet de lire tout le contenu du fichier. Cette méthode retourne une chaine de caractères représentant tout le contenu du fichier.
  - **f.read(n) :** permet de lire n caractères.
  - **f.readlines() :** permet de lire tout le contenu du fichier. Cette méthode retourne une liste composée de chaines de caractères. Chaque chaine représente une ligne du fichier.
  - **f.readline() :** permet de lire une ligne à la fois. Cette méthode retourne une chaine de caractères représentant la ligne courante lue.

Écriture

- Pour écrire dans un fichier, on utilise l'une des méthodes suivantes :
  - **f.write('texte') :** permet d'écrire la chaine de caractères ('texte') passée en paramètre.
  - **f.writelines(L) :** permet d'écrire une liste L, où chaque élément de L est une chaine de caractères.

NB

- Pour constituer des lignes dans le fichier, on doit ajouter le symbole " \n" (symbole retour chariot).

Le module os

- Par défaut, l'interprète PYTHON effectue les opérations d'ouverture/lecture/écriture à partir du répertoire courant. Il se peut que le fichier avec lequel vous travaillez soit placé dans un autre répertoire. Pour cela, on peut changer de répertoire de travail.

➡ Utiliser le module **os.**

- L'import du module os :

import os as os

- Pour connaitre le répertoire courant :

os.getcwd()

'/home/saloua'

- Pour changer de répertoire courant :

os.chdir('/home/lsaloua/Bureau/Cours_Programmation')

Le module pickle

- Le module pickle permet d'enregistrer des objets de n'importe quel type en mode binaire. L'avantage est que ces objets peuvent être récupérés facilement par la suite.
- Pour importer le module pickle :

import pickle as p

Pour enregistrer un objet (ici un dictionnaire) :

f=open('lename', 'wb')

obj = {1 : 'a' , 2 : 'b'}

p.dump(obj, f)

- Pour récupérer l'objet :

f=open('lename', 'rb')

obj = p.load(f)

Exemple :

```
import pickle as p
# creation d'objets de différents types
etudiants= { 1 : ' Ali ' , 2 : ' Khaoula ' , 3 : ' saloua' }
L=[ ' Bonjour ' ,123 , ' abc ' ]
T=(L , ' soleil ' )
a=12
b=12.5
# enregistrement des objets
f = open ( 'mes_donnees.bin ' , 'wb ' )
p . dump( etudiants , f )
p . dump(L , f )
p . dump(T, f )
p . dump( a , f )
p . dump( b , f )
f . close ( )
```

```
# recupération des objets
f = open ( 'mes_donnees.bin ' , ' rb ' )
o1=p .load( f )
o2=p . load( f )
o3=p . l o a d ( f )
o4=p . load( f )
o5=p . load( f )
print ( o1 )
print ( o2 )
print ( o3 )
print ( o4 )
print ( o5 )
f . close ( )
```

CHAPITRE 2

Manipuler les fichiers logs

Ce que vous allez apprendre dans ce chapitre :

- Ecriture de scripts pour la création de logs (scripts avancés)
- Test/exécution des scripts

01 heure

CHAPITRE 2

Manipuler les fichiers logs

1. **Ecriture de scripts pour la création de logs (scripts avancés)**
2. Test/exécution des scripts

Debugger un programme.

- Utilisation souvent de **print**.

➡ N'est pas satisfaisant :

  - Les informations sont envoyées sur la sortie standard.
  - Mélange d'informations liées au fonctionnement du programme et l'utilisation classique de la sortie standard.
  - Il n'y a pas forcement de sortie standard (programme Python utilisé comme démon).
  - Les messages ne sont pas formatés.

Logging

- Logging est un framework standard de Python.
- Similaire à Log4j du monde Java.
- Permet de formater les messages.
- Permet de conditionner les messages.
- Permet d'envoyer les messages sur un ou plusieurs flux (handler).

Principe de fonctionnement (voir figure p. 151)

- Principe de fonctionnement très simple : à la demande, certaines informations sont stockées dans un fichier texte, notre fichier de **log**.
- Définir des niveaux de criticité qui indiquent l'importance du message. Il peut s'agir d'un simple message informatif ou d'un message très important.
- Le but est de tracer l'activité de notre programme afin de pouvoir analyser son exécution en cas d'anomalie.

Les différents niveaux de log

- Les logs sont répartis en différents niveaux.
- Chaque niveau correspond à une importance donnée.
- Par défaut cinq niveaux sont prédéfinis :
    - **DEBUG :** débogage, tracer des informations détaillées.

Exemple : valeur de variables

    - **INFO :** niveau d'information standard, tracer les événements classiques.

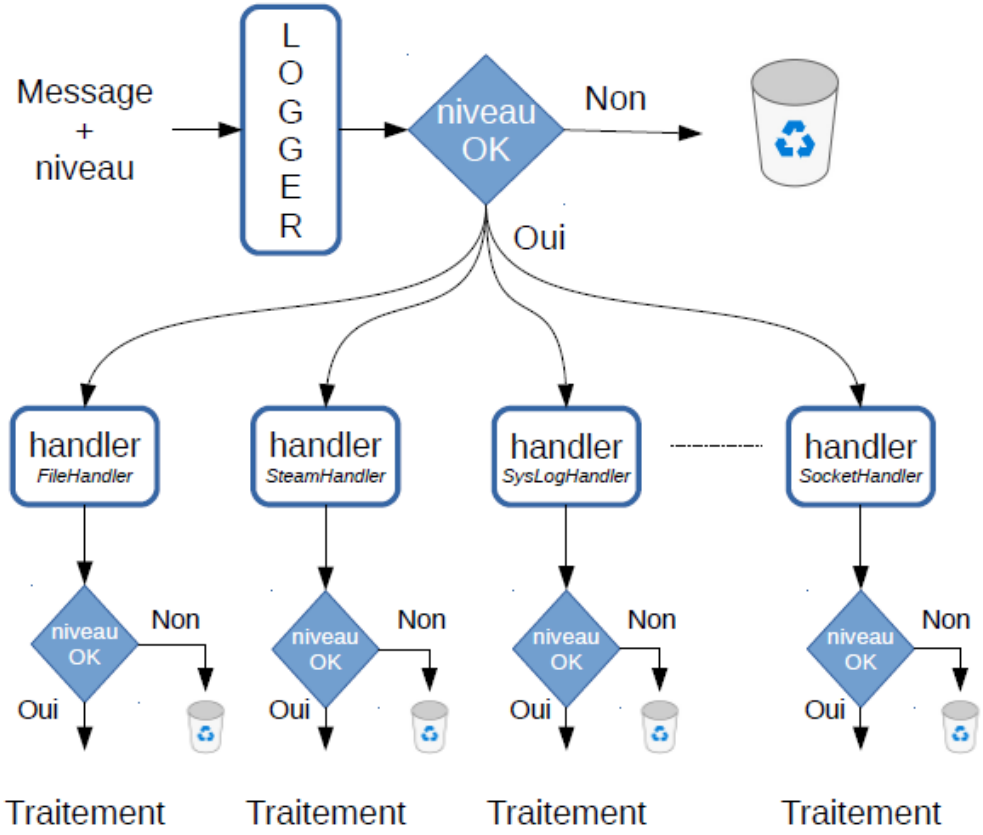
Exemple : le lancement d'une fonction, transfert d'un fichier…

    - **WARNING :** un événement qui pourrait avoir son importance.

Exemple : seuil atteint

    - **ERROR :** erreur sévère, mais n'ayant pas entraîné le crash du programme.
    - **CRITICAL :** une erreur ayant entraîné le plantage de l'application.

Figure (principe de fonctionnement)



- inspiré de http://sametmax.com/ecrire-des-logs-en-python/

Utilisation d'un logger dans votre code

1. Importer le module logging.
2. Obtenir le logger : fonction getLogger.
3. Utiliser la méthode :
    - log avec comme premier paramètre effectif le niveau de log et, comme deuxième, le message.
    - **debug, info, warning, error** ou **critical** avec comme paramètre effectif le message.

Paramétrage du logger et création des handler

1. Importer le module logging ;
2. Obtenir le logger : fonction getLogger ;
3. Fixer le niveau de logger : méthode setLevel avec les constantes du module logging :
4. Créer un handler, plusieurs classes disponibles : StreamHandler (sortie erreur standard), FileHandler (fichier de log), etc. ;
5. Paramétrer le handler : setLevel (niveau), setFormatter (formatage), etc. ;
6. Ajouter le handler au logger : méthode addHandler.

CHAPITRE 2

Manipuler les fichiers logs

1. **Ecriture de scripts pour la création de logs (scripts avancés)**
2. Test/exécution des scripts

Exemple :

```
1. import logging
2. logging.basicConfig(filename='test_log.log',level=logging.DEBUG,\
3. format='%(asctime)s -- %(name)s -- %(levelname)s -- %(message)s')
4.  logging.debug('Debug error')
5. logging.info('INFO ERROR')
6. logging.warning('Warning Error %s: %s', '01234', 'Erreur Oracle')
7. logging.error('error message')
8. logging.critical('critical error')
```

Explication du code

- Ligne 1, nous importons notre module.
- Ligne 2, nous définissons notre fichier de log. Nous passons tout d'abord le filename, en chemin relatif (ici au même niveau que notre script) ou absolu. Ensuite, nous définissons le niveau de sensibilité, puis enfin le format de sortie.
- Ligne 4, 5, 6, 7, et 8 vous voyez comment saisir des entrées dans le fichier de log. C'est au développeur de définir l'importance de l'information. Le seul argument à passer est alors le message à écrire, une simple chaine de caractères.

Parser un fichier de log

- Lire le fichier ligne par ligne.
- Fractionner le contenu de cette ligne avec le caractère de séparation choisi (dans notre exemple précédent, « -- »).

Exemple :

- with open("test_log.log", "r") as log_file:
- for line in log_file.readlines():
- tmp = line.split("--") print("time: %s, user: %s, level: %s, message: %s" % (tmp[0], tmp[1], tmp[2], tmp[3]))

Outils de débogage Python

- Module **pdb** dans la bibliothèque Python.
- **Winpdb** (http://winpdb.org/), un débogueur graphique (aucun lien avec le Windows de Microsoft !).
- **PuDB** (http://pypi.python.org/pypi/pudb), un débogueur sémi-graphique.
- **pydbgr** (http://code.google.com/p/pydbgr/), une version améliorée de pdb.

Environnements de développement avec débogueur :

- IDLE (fait partie de la distribution Python).
- Komodo IDE (http://www.activestate.com/komodo/features/).
- WingIDE (http://wingware.com/).
- PyDev (http://pydev.org/), une extension pour Eclipse.

CHAPITRE 3

Tester le fonctionnement des scripts

Ce que vous allez apprendre dans ce chapitre :

- Collecte des résultats retournés par le script
- Évaluation des cas de test
- Déploiement des scripts

01 heure

CHAPITRE 3

Tester le fonctionnement des scripts

1. **Collecte des résultats retournés par le script**
2. Évaluation des cas de test
3. Déploiement des scripts

unittest - Unit testing framework

- Le framework de tests unitaires unittest a été inspiré à l'origine par JUnit et a une saveur similaire aux principaux frameworks de tests unitaires dans d'autres langages. Il prend en charge l'automatisation des tests, le partage du code de configuration et d'arrêt des tests, l'agrégation des tests dans des collections et l'indépendance des tests par rapport au cadre de création de rapports.
- Pour y parvenir, unittest prend en charge certains concepts importants de manière orientée objet :
    - **test fixture :** Un montage de test représente la préparation nécessaire pour effectuer un ou plusieurs tests, et toutes les actions de nettoyage associées. Cela peut impliquer, par exemple, la création de bases de données temporaires ou proxy, de répertoires ou le démarrage d'un processus serveur.
    - **test case :** Un cas de test est l'unité individuelle de test. Il vérifie une réponse spécifique à un ensemble particulier d'entrées. unittest fournit une classe de base, TestCase, qui peut être utilisée pour créer de nouveaux cas de test.
    - **test suite :** Une suite de tests est un ensemble de cas de test, de suites de tests ou des deux. Il est utilisé pour agréger les tests qui doivent être exécutés ensemble.
    - **test runner :** Un exécuteur de test est un composant qui orchestre l'exécution des tests et fournit le résultat à l'utilisateur. L'exécuteur peut utiliser une interface graphique, une interface textuelle ou renvoyer une valeur spéciale pour indiquer les résultats de l'exécution des tests.

Groupement de test

- **class unittest.TestSuite(tests=()):**
- Cette classe représente une agrégation de cas de test individuels et de suites de tests. La classe présente l'interface requise par le lanceur de test pour lui permettre d'être exécuté comme n'importe quel autre cas de test. L'exécution d'une instance TestSuite revient à itérer sur la suite, en exécutant chaque test individuellement.
- Si des tests sont donnés, il doit s'agir d'un itérable de cas de test individuels ou d'autres suites de tests qui seront utilisés pour construire la suite initialement. Des méthodes supplémentaires sont fournies pour ajouter ultérieurement des cas de test et des suites à la collection.
- Les objets TestSuite se comportent un peu comme les objets TestCase, sauf qu'ils n'implémentent pas réellement de test. Au lieu de cela, ils sont utilisés pour agréger les tests en groupes de tests qui doivent être exécutés ensemble
- Certaines méthodes supplémentaires sont disponibles pour ajouter des tests aux instances TestSuite :
    - addTest(test)
    - Ajoutez un TestCase ou TestSuite à la suite.
    - addTests(tests)
- Ajoutez tous les tests d'un itérable d'instances TestCase et TestSuite à cette suite de tests. Cela équivaut à itérer sur les tests, en appelant addTest() pour chaque élément.
    - TestSuite partage les méthodes suivantes avec TestCase :
    - run(result)
- Exécutez les tests associés à cette suite, en collectant le résultat dans l'objet de résultat de test transmis comme résultat. Notez que contrairement à TestCase.run(), TestSuite.run() nécessite que l'objet de résultat soit transmis.
    - debug()
- Exécutez les tests associés à cette suite sans collecter le résultat. Cela permet aux exceptions déclenchées par le test d'être propagées à l'appelant et peut être utilisée pour prendre en charge l'exécution de tests sous un débogueur.
    - countTestCases()
- Renvoie le nombre de tests représentés par cet objet de test, y compris tous les tests individuels et les sous-suites.

CHAPITRE 3

Tester le fonctionnement des scripts

1. Collecte des résultats retournés par le script
2. **Évaluation des cas de test**
3. Déploiement des scripts

Cas de test

- Utilisation de la bibliothèque standard Python pour créer les scripts de test unitaire
- **unittest** est une bibliothèque standard Python qui prend en charge le concept orienté objet. Elle a les caractéristiques suivantes :
    - **text fixture**: Un text fixture représente la préparation nécessaire pour effectuer un ou plusieurs tests et toutes les actions de nettoyage associées. Cela peut impliquer, par exemple, la création de bases de données temporaires ou proxy, de répertoires ou le démarrage d'un processus serveur.
    - **test case** : Un cas de test est l'unité individuelle de test. Il vérifie une réponse spécifique à un ensemble particulier d'entrées. unittest fournit une classe de base, TestCase, qui peut être utilisée pour créer de nouveaux cas de test.
    - **test suite**: Un test suite est un ensemble de cas de test, de suites de tests ou des deux. Il est utilisé pour agréger les tests qui doivent être exécutés ensemble.
    - **test runner**: Un test runner est un composant qui orchestre l'exécution des tests et fournit le résultat à l'utilisateur. L'exécuteur peut utiliser une interface graphique, une interface textuelle ou renvoyer une valeur spéciale pour indiquer les résultats de l'exécution des tests.

Dans ce cours, nous nous concentrerons sur le cas de test (test case). Commençons par l'écriture du code en Python.

Cas de test

- **Import**

Tout d'abord, vous devez importer le module en le déclarant au-dessus de votre fichier Python : **import unittest**

- **TestCase**

Pour créer un cas de test, vous devez créer une classe qui hérite de la classe unittest.TestCase: **class TestFunctions(unittest.TestCase):**

- **Méthodes de test**

Une méthode de test peut être créée en les préfixant avec le mot-clé **test**. Toute fonction commençant par le mot-clé test sera traitée comme une méthode de test unitaire: **def test_some_function(self):**

Exemple:

- Une fonction simple qui supprime les espaces de la chaîne d'entrée. Il accepte une chaîne d'entrée et affiche le même texte sans l'espace blanc.

```
def remove_whitespace(text):
    result = text.replace(' ', '')
    return result
```

Cas de test

- Ensuite, nous devons créer une méthode de test qui teste la fonction que nous avons définie ci-dessus. Nous utilisons simplement le même nom et le préfixe avec le mot-clé test. N'hésitez pas à lui donner le nom que vous voulez tant qu'il commence par le mot-clé test. Dans la fonction, vous devez définir la vérification du résultat attendu à l'aide de l'appel assertEqual.

```
def test_remove_whitespace(self):
    self.assertEqual(remove_whitespace('今天 我 过得 很 开心'), '今天我过得很开心')
    self.assertEqual(remove_whitespace('これ は 俺 の 本 です'), 'これは俺の本です')
    self.assertEqual(remove_whitespace('Welcome to Medium'), 'WelcometoMedium')
```

- Dans l'exemple donné ci-dessus, nous avons citer trois lignes qui vérifient si la sortie de la fonction remove_whitespace est la même que le résultat attendu. Dans un cas d'utilisation réel, vous devez couvrir toutes les possibilités auxquelles vous pouvez penser.

Cas de test

- **Main**

Une fois que vous avez terminé, écrivez le code suivant pour terminer le script de votre premier test unitaire :

```
if __name__ == '__main__':
    unittest.main()
```

- Enregistrez le script en tant que fichier Python et exécutez-le en utilisant la syntaxe suivante dans l'invite de commande:

```
python -m unittest myFile.py
```

CHAPITRE 3

Tester le fonctionnement des scripts

1. Collecte des résultats retournés par le script
2. Évaluation des cas de test
3. **Déploiement des scripts**

Déploiement des scripts

Dans ce que suit, vous apprendrez à déployer et à exécuter un script Python sur une machine Windows 10, 64 bits sur laquelle Python n'est pas installé.

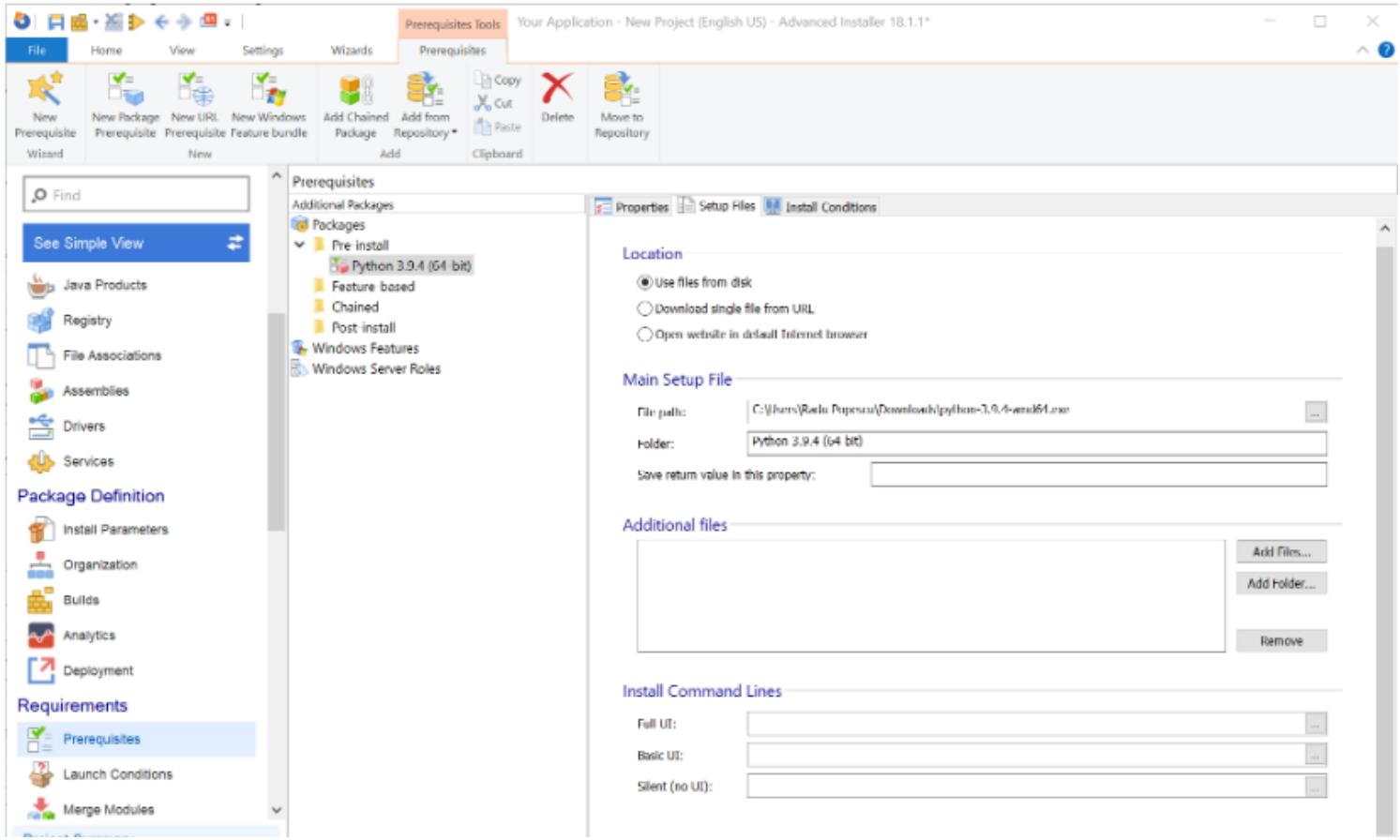
- **Ajout du script Python :**

Nous supposerons que nous voulons déployer uniquement sur une machine 64 bits et exécuter le programme d'installation Python en mode silencieux.

1. Téléchargez la dernière version de Python à partir du site Web. Nous utiliserons le programme d'installation exécutable Windows 64 bits ;
2. Accédez à la page Prérequis de votre projet Advanced Installer ;
3. Sélectionnez la section Pré-installation afin que votre prérequis s'installe au cours de cette étape ;
4. Appuyez sur "New Package Prerequisite" dans la barre supérieure ;
5. Sélectionnez le programme d'installation exécutable Python sur votre disque et appuyez sur "Ouvrir« ;
6. Accédez à l'onglet "Setup Files" de votre prérequis nouvellement ajouté.

PARTIE 5

Déploiement des scripts



Déploiement des scripts

7. Écrivez le code ci-dessous dans le champ "Full UI", afin que le programme d'installation de Python s'exécute en mode silencieux.

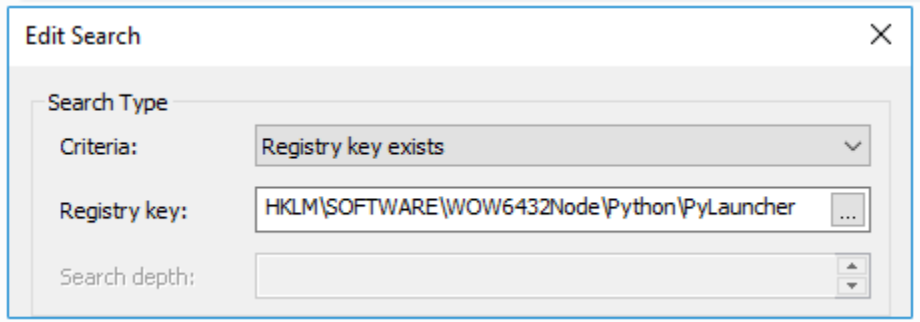
/quiet InstallAllUsers=1 PrependPath=1 Include_test=0

COPY

8. Allez dans «Install Conditions” -> “Supported Windows Versions » -> « Versions Windows prises en charge », décochez « Versions Windows 32 bits ».
9. Développez les "64-bit Windows versions" et vérifiez la version Windows que vous souhaitez cibler.
10. Dans la section « Install Conditions », activez « Install prerequisite based on conditions ».
11. Créez un nouveau critère en appuyant New ->Criteria -> Registry key exists.
12. Inscrivez dans le champ «Registry key», la clé de registre du lanceur Python ; cela vérifiera avec PyLauncher existe sur le PC et installera ou n'installera pas le prérequis en fonction du résultat.

HKLM\SOFTWARE\WOW6432Node\Python\PyLauncher

COPY



13. Supprimez le critère «File version» de la liste.

PARTIE 5

Déploiement des scripts

- **Vérification du prérequis :**

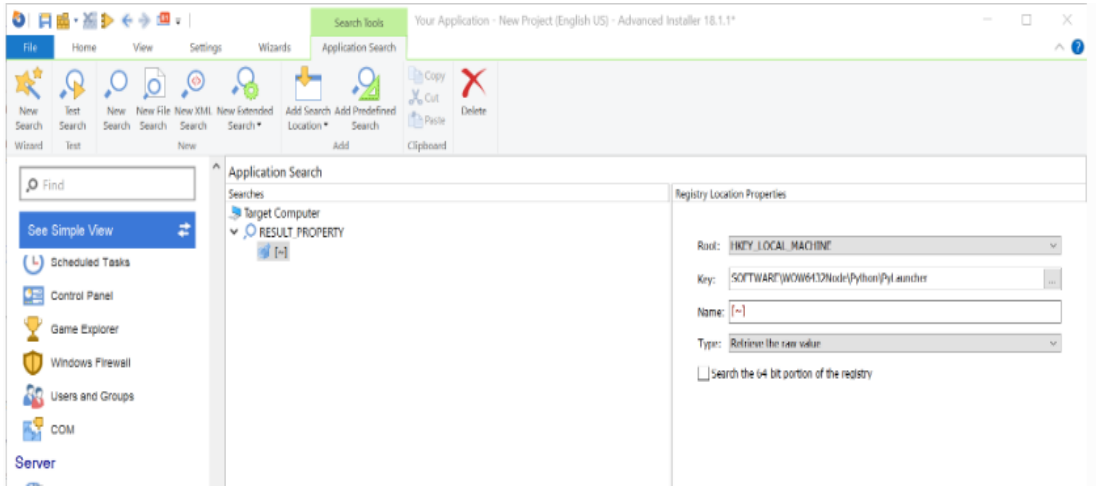
Nous voulons nous assurer que le prérequis a été installé correctement avant d'exécuter notre action personnalisée.

1. Accédez à la page de recherche ;
2. Appuyez sur Nouvelle recherche dans la barre supérieure et suivez l'assistant de recherche ;
3. Sélectionnez « Search for a registry value » ;
4. Écrivez dans la "Key" la valeur ci-dessous, pour vérifier si le lanceur Python existe.

SOFTWARE\WOW6432Node\Python\PyLauncher

COPY

5. Écrivez [~] dans le champ « Value »
6. Laissez l'interprétation à «Retrieve the raw value».



Cochez "Test your search now" et appuyez sur Finish.

Si le lanceur Python est installé, le résultat de la recherche est "C:\WINDOWS\py.exe".

PARTIE 5

Déploiement des scripts

- **Exécuter le script Python :**

Nous voulons exécuter le script Python à la fin de l'installation. Pour y parvenir, nous utiliserons une action personnalisée PowerShell qui lancera votre script.

1. Accédez à la page Custom Actions page ;
2. Créez une action personnalisée «Run PowerShell inline script» ;
3. Déplacez-le sous l'étape "Finish Execution" ; cela transformera le "Execution Time" en Immédiatement ;
4. Ajoutez les commandes suivantes dans votre action personnalisée :

$scriptPath = AI_GetMsiProperty APPDIR

$scriptPath = $scriptPath + "HelloWorld.py"

start-process $scriptPath

COPY

Déploiement des scripts

Le code définira la variable $scriptPath sur la valeur APPDIR (dossier Application) et la concaténera avec le nom de votre script. Ensuite, il exécutera le fichier défini dans la variable $scriptPath.

5. Ne laissez activée que l'option "Installer" dans la condition d'étape d'exécution, afin que votre script ne s'exécute que pendant cette étape.

6. Définissez la "Condition" sur la valeur ci-dessous, pour vérifier si "RESULT_PROPERTY" contient le chemin correct pour le lanceur Python, avant de lancer votre script Python.

RESULT_PROPERTY = "C:\WINDOWS\py.exe"

COPY