Programmation en Python

Cours Système - HLIN303 / HMIN113M

Pierre Pompidor

Introduction - Python vs les autres langages de script

Python est un langage ubiquiste pour

- le prototypage d'applications
- l'extension d'applications
- le calcul scientifique
- **le scripting système**
- la fouille de données (data mining)
- ...

Python est un langage modulaire

Introduction - Python vs les autres langages de script

Python est un langage ubiquiste pour

- le prototypage d'applications
- l'extension d'applications
- le calcul scientifique
- **le scripting système**
- la fouille de données (data mining)
- ...

Python est un langage modulaire

Python est un langage efficace

Introduction - Python vs les autres langages de script

Python est un langage ubiquiste pour

- le prototypage d'applications
- l'extension d'applications
- le calcul scientifique
- **le scripting système**
- la fouille de données (data mining)
- ...

Python est un langage modulaire

Python est un langage efficace

Python est un langage pédagogique

Introduction - Python est un langage interprété

L'interpréteur python : python3

- traduit chaque instruction en langage binaire
- il n'y a pas d'exécutable stocké sur votre système de fichiers (contrairement par exemple à un programme écrit avec le langage C ou C++)

Introduction - Python est un langage interprété

L'interpréteur python : python3

- traduit chaque instruction en langage binaire
- il n'y a pas d'exécutable stocké sur votre système de fichiers (contrairement par exemple à un programme écrit avec le langage C ou C++)

Il est possible de programmer

- directement sous l'intepréteur (python3)
- en créant des scripts

Introduction - Python est modulaire

Notion de modules

- Un module est un fichier contenant du code python
- Un module permet de ne charger en mémoire que ce que l'on a besoin
- Ainsi du code peut-être réemployé dans une autre application

Introduction - Python est modulaire

Notion de modules

- Un module est un fichier contenant du code python
- Un module permet de ne charger en mémoire que ce que l'on a besoin
- Ainsi du code peut-être réemployé dans une autre application

Utilisation d'un module

import <nomModule>

Introduction - Python est modulaire

Notion de modules

- Un module est un fichier contenant du code python
- Un module permet de ne charger en mémoire que ce que l'on a besoin
- Ainsi du code peut-être réemployé dans une autre application

Utilisation d'un module

import <nomModule>

Installation d'un module "public"

pip install <nomModule>

Introduction - Un premier exemple de script python

bonjour.py

```
nom = input("Quel est ton nom ?")
print("Bonjour", nom)
```

Introduction - Un premier exemple de script python

bonjour.py

```
nom = input("Quel est ton nom ?")
print("Bonjour", nom)
```

Ajout du droit d'exécution dans un shell (terminal Linux)

chmod +x bonjour.py

Introduction - Un premier exemple de script python

bonjour.py

```
nom = input("Quel est ton nom ?")
print("Bonjour", nom)
```

Ajout du droit d'exécution dans un shell (terminal Linux)

```
chmod +x bonjour.py
```

Exécution du script dans un shell (terminal Linux)

```
python3 ./bonjour.py
```

Introduction - exécution d'un script Python

Gestion des droits (par exemple rwxr-xr–)

```
chmod 755 <nomDuScript>
chmod +x <nomDuScript>
```

Introduction - exécution d'un script Python

Gestion des droits (par exemple rwxr-xr–)

```
chmod 755 <nomDuScript>
chmod +x <nomDuScript>
```

9 caractères répartis en trois groupes :

- 3 populations : propriétaire, le groupe, les autres
- 3 types de droits : r (read), w (write), x (execute ou access)

Introduction - exécution d'un script Python

Gestion des droits (par exemple rwxr-xr–)

```
chmod 755 <nomDuScript>
chmod +x <nomDuScript>
```

9 caractères répartis en trois groupes :

- 3 populations : propriétaire, le groupe, les autres
- 3 types de droits : r (read), w (write), x (execute ou access)

Utilisation d'un éditeur de texte (par exemple emacs)

```
emacs <nomDuScript> &
```

Introduction - exécution d'un script Python

Gestion des droits (par exemple rwxr-xr–)

```
chmod 755 <nomDuScript>
chmod +x <nomDuScript>
```

9 caractères répartis en trois groupes :

- 3 populations : propriétaire, le groupe, les autres
- 3 types de droits : r (read), w (write), x (execute ou access)

Utilisation d'un éditeur de texte (par exemple emacs)

```
emacs <nomDuScript> &
```

Le & permet de laisser le terminal actif

Trois exercices de TD/TP extrêmement agréables

Filtrage d'informations prises dans un fichier

- Recherche des capitales d'un certain nombre de pays : création d'un petit quizz

Trois exercices de TD/TP extrêmement agréables

Filtrage d'informations prises dans un fichier

- Recherche des capitales d'un certain nombre de pays : création d'un petit quizz

Analyse d'une arborescence de dossiers

- Parcours récursif d'un dossier d'accueil et production d'un fichier récapitulatif par types de fichiers rencontrés

Trois exercices de TD/TP extrêmement agréables

Filtrage d'informations prises dans un fichier

- Recherche des capitales d'un certain nombre de pays : création d'un petit quizz

Analyse d'une arborescence de dossiers

- Parcours récursif d'un dossier d'accueil et production d'un fichier récapitulatif par types de fichiers rencontrés

Analyse du résultat d'un exécutable

- Vue synthétique (reporting) du résultat d'une commande système (par exemple last)

Structures de données de base

Une variable correspond à l'allocation d'une zone de la mémoire vive

Les variables scalaires

```
i = 1
f = 1.0
chaine = "Bonjour"
booleen = False
```

Structures de données de base

Une variable correspond à l'allocation d'une zone de la mémoire vive

Les variables scalaires

```
i = 1
f = 1.0
chaine = "Bonjour"
booleen = False
```

Le typage est dynamique

- Le type de la variable n'est pas précisé lors de sa création
- Mais le transtypage n'est pas automatique

```
i = int("123")
```

Structures de données de base - suite

Allocation/désallocation automatique de la mémoire

- La mémoire nécessaire est automatiquement allouée
- Et cela permet d'éviter le principal risque de bug

Structures de données de base - suite

Allocation/désallocation automatique de la mémoire

- La mémoire nécessaire est automatiquement allouée
- Et cela permet d'éviter le principal risque de bug

Erreur fréquente en C

```
char* chaine = "Bonjour";
// il y a deja un risque
strcat(chaine, " toi"); // second risque
   // (meme si cela semble fonctionner)
```

Structures de données de base - suite

Allocation/désallocation automatique de la mémoire

- La mémoire nécessaire est automatiquement allouée
- Et cela permet d'éviter le principal risque de bug

Erreur fréquente en C

```
char* chaine = "Bonjour";
// il y a deja un risque
strcat(chaine, " toi"); // second risque
   // (meme si cela semble fonctionner)
```

Pas de souci en python

```
chaine = "Bonjour"
chaine += " toi"
```

Structures de données de base - les chaînes (strings)

Exemple de création de chaînes de caractères

```
chaine1 = "Bonjour"
chaine2 = 'Hello'
chaine3 = "Aujourd'hui"
```

Structures de données de base - les chaînes (strings)

Exemple de création de chaînes de caractères

```
chaine1 = "Bonjour"
chaine2 = 'Hello'
chaine3 = "Aujourd'hui"
```

Nombre de caractères d'une chaîne : len()

```
print(len(chaine3))   # Affiche : 11
```

Structures de données - les listes

Les listes sont des tableaux dynamiques

```
menu = ['entree', 'plat', 'dessert']
print(menu[1])   # affiche : plat
```

Structures de données - les listes

Les listes sont des tableaux dynamiques

```
menu = ['entree', 'plat', 'dessert']
print(menu[1])   # affiche : plat
```

Syntaxe

- Les crochets encadrent les éléments de la liste
- Les éléments sont indicés par un entier (0 pour le pr. élément)
- Un élément d'une liste peut être de n'importe quelle valeur

Structures de données - les listes

Les listes sont des tableaux dynamiques

```
menu = ['entree', 'plat', 'dessert']
print(menu[1])   # affiche : plat
```

Syntaxe

- Les crochets encadrent les éléments de la liste
- Les éléments sont indicés par un entier (0 pour le pr. élément)
- Un élément d'une liste peut être de n'importe quelle valeur

Une liste peut-être initialisée à vide

```
listeVide = []
```

Structures de données - les listes

Opérations sur les listes

```
liste = [1, 2, 3]
liste.append("partez␣!")  # ajoute un element
del liste[2]  # supprime le 3ieme element
liste.remove(1)  # suppr. le 1er elt de valeur 1
```

Structures de données - les listes

Opérations sur les listes

```
liste = [1, 2, 3]
liste.append("partez␣!")  # ajoute un element
del liste[2]  # supprime le 3ieme element
liste.remove(1)  # suppr. le 1er elt de valeur 1
```

Ajout et suppression d'éléments dans une liste

- append() et remove() : méthodes sur l'objet liste
- del est une pseudo-fonction

Structures de données - les listes

Opérations sur les listes

```
liste = [1, 2, 3]
liste.append("partez␣!")  # ajoute un element
del liste[2]  # supprime le 3ieme element
liste.remove(1)  # suppr. le 1er elt de valeur 1
```

Ajout et suppression d'éléments dans une liste

- append() et remove() : méthodes sur l'objet liste
- del est une pseudo-fonction

Tranches de listes (slices)

```
liste = [1, 2, 3, "partez", "!"]
print(liste[1:4])  # [1,4[
                   # affiche [2, 3, "partez"]
```

Structures de données - La liste des paramètres du script

Il faut importer le module sys

Structures de données - La liste des paramètres du script

Il faut importer le module sys

La liste des paramètres est sys.argv

Structures de données - La liste des paramètres du script

Il faut importer le module sys

La liste des paramètres est sys.argv

Soit le script bonjour.py

```
import sys

print(sys.argv[0])
print("Bonjour", sys.argv[1])
```

Structures de données - La liste des paramètres du script

Il faut importer le module sys

La liste des paramètres est sys.argv

Soit le script bonjour.py

```
import sys

print(sys.argv[0])
print("Bonjour", sys.argv[1])
```

./bonjour.py Pierre

```
./bonjour.py
Bonjour Pierre
```

Structures de données - les listes et les tuples

Les listes peuvent elles-mêmes contenir des listes (ou d'autres structures de données)

```
liste = [1, 2, 3, ["partez", "maintenant"]]
print(liste[3][1])   # affiche : maintenant
```

Structures de données - les listes et les tuples

Les listes peuvent elles-mêmes contenir des listes (ou d'autres structures de données)

```
liste = [1, 2, 3, ["partez", "maintenant"]]
print(liste[3][1])   # affiche : maintenant
```

Les tuples : des listes constantes (non modifiables)

- Les tuples sont encadrés par des parenthèses
- Mais l'accès aux éléments se fait toujours via les crochets

Structures de données - les listes et les tuples

Les listes peuvent elles-mêmes contenir des listes (ou d'autres structures de données)

```
liste = [1, 2, 3, ["partez", "maintenant"]]
print(liste[3][1])   # affiche : maintenant
```

Les tuples : des listes constantes (non modifiables)

- Les tuples sont encadrés par des parenthèses
- Mais l'accès aux éléments se fait toujours via les crochets

Un exemple de tuple

```
unTuple = (1, 2, 3)
print(unTuple[0])
unTuple[0] = "un"   # cela provoque une erreur !
```

Opérations sur les listes - suite

```
liste = [1, 2, 3, "partez", "!"]
nouvelleListe = liste[:]
```

Opérations sur les listes - suite

```
liste = [1, 2, 3, "partez", "!"]
nouvelleListe = liste[:]
```

Copie de listes en profondeur

```
nouvelleListe = copy.deepcopy(liste)
# copie les objets ou les listes inserees
```

Opérations sur les listes - suite

```
liste = [1, 2, 3, "partez", "!"]
nouvelleListe = liste[:]
```

Copie de listes en profondeur

```
nouvelleListe = copy.deepcopy(liste)
# copie les objets ou les listes inserees
```

Affichage des éléments d'une liste (ci-après)

- par une boucle avec indice
- par une boucle automatique

Structures de base - le bloc d'instructions

Exemple de bloc d'instruction assujetti à un test

```
if <expression conditionnelle> :
    instruction 1
    instruction 2
    ...
    instruction n
instruction hors du bloc (car alignee sur le if)
```

Structures de base - le bloc d'instructions

Exemple de bloc d'instruction assujetti à un test

```
if <expression conditionnelle> :
    instruction 1
    instruction 2
    ...
    instruction n
instruction hors du bloc (car alignee sur le if)
```

Syntaxe d'un bloc d'instructions

- Le bloc d'instructions est initié par un :
- Les instructions sont indentées
  ne mélangez pas des espaces avec des indentations

Structures de base - La structure conditionnelle

Exemple de structure conditionnelle

```
import sys
if len(sys.argv) != 2 :
    print("Le␣script␣doit␣avoir␣un␣parametre")
    exit()
```

Structures de base - La structure conditionnelle

Exemple de structure conditionnelle

```
import sys
if len(sys.argv) != 2 :
    print("Le␣script␣doit␣avoir␣un␣parametre")
    exit()
```

Le "sinon" est introduit par else

```
if <expression conditionnelle> :
    premiere instruction du bloc
    ...
else :
    premiere instruction du bloc
    ...
```

Structures de base - La structure conditionnelle

Exemple de structures conditionnelle

```
if len(liste) > 0 and len(liste) < 10 :
```

Structures de base - La structure conditionnelle

Exemple de structures conditionnelle

```
if len(liste) > 0 and len(liste) < 10 :
```

Opérateurs

- opérateurs logiques : and or
- opérateurs de comparaison classiques
  attention : == et !=
- une expression renvoie faux si l'évaluation est égale à 0 ou ""

Exemple d'une boucle while avec indice de boucle

```
liste = [1, 2, 3, "partez", "!"]
i = 0
while i < len(liste) :
        print(liste[i])
        i += 1
```

Structures de base - Les structures de boucle

Exemple d'une boucle while avec indice de boucle

```
liste = [1, 2, 3, "partez", "!"]
i = 0
while i < len(liste) :
        print(liste[i])
        i += 1
```

Syntaxe

```
<initialisation de l'indice de boucle>
while <expression conditionnelle> :
      instructions du bloc
      <(in|de)crement de l'indice du boucle>
```

Exemple d'une boucle for automatique

```
liste = [1, 2, 3, "partez", "!"]
for valeur in liste :
    print(valeur)
```

Structures de base - Les structures de boucle

Exemple d'une boucle for automatique

```
liste = [1, 2, 3, "partez", "!"]
for valeur in liste :
    print(valeur)
```

Syntaxe

```
for <variable locale> in <liste> :
    instructions du bloc
```

Les compréhensions de listes

Une syntaxe compacte pour initialiser une liste

Exemple de compréhension de liste

```
liste = [i*2 for i in range(10)]
for e in liste :
        print(e)
```

Les compréhensions de listes

Une syntaxe compacte pour initialiser une liste

Exemple de compréhension de liste

```
liste = [i*2 for i in range(10)]
for e in liste :
        print(e)
```

Fonction range()

- range() permet de générer une liste d'entiers :
  range(10) génère [0,1,2,3,4,5,6,7,8,9]

Paramètres, test, boucle, création d'une liste

- Vérifiez que le script possède au moins un paramètre
- Affichez le premier paramètre passé au script
- Affichez tous les paramètres
- Affichez tous les paramètres sauf le nom du script

Cours Introductif - Exercices de TD

Paramètres, test, boucle, création d'une liste

- Vérifiez que le script possède au moins un paramètre
- Affichez le premier paramètre passé au script
- Affichez tous les paramètres
- Affichez tous les paramètres sauf le nom du script

Paramètres, test, boucle, création d'une liste

- Calculez en itératif la factorielle de n (n est donné en paramètre)
- Affichez les n premiers nombres premiers (n est donné en paramètre)

Structures de données - Les dictionnaires

Exemple de dictionnaire

```
nbJoursMois = {'janvier':31, 'fevrier':28.25 }
print(nbJoursMois['fevrier'])   # Affiche : 28.25
```

Structures de données - Les dictionnaires

Exemple de dictionnaire

```
nbJoursMois = {'janvier':31, 'fevrier':28.25 }
print(nbJoursMois['fevrier'])   # Affiche : 28.25
```

Les dictionnaires permettent :

- de rajouter de la sémantique (les éléments sont nommés par une clef)
- d'accéder directement à l'emplacement de l'élément en mémoire
  (généralement via une fonction de hashage)

Structures de données - Les dictionnaires

Exemple de dictionnaire

```
nbJoursMois = {'janvier':31, 'fevrier':28.25 }
print(nbJoursMois['fevrier'])   # Affiche : 28.25
```

Les dictionnaires permettent :

- de rajouter de la sémantique (les éléments sont nommés par une clef)
- d'accéder directement à l'emplacement de l'élément en mémoire
  (généralement via une fonction de hashage)

Un dictionnaire peut-être initialisé à vide

```
dicoVide = {}
```

Structures de données - Les dictionnaires - suite

Affichage des éléments d'un dictionnaire

```
nbJoursMois = {'janvier':31, 'fevrier':28.25,
               'mars':31, ...}
for clef in nbJoursMois :
    print(clef, ":", nbJoursMois[clef])
for (clef, valeur) in items(nbJoursMois) :
    print(clef, ":", valeur)
```

Structures de données - Les dictionnaires - suite

Affichage des éléments d'un dictionnaire

```
nbJoursMois = {'janvier':31, 'fevrier':28.25,
               'mars':31, ...}
for clef in nbJoursMois :
    print(clef, ":", nbJoursMois[clef])
for (clef, valeur) in items(nbJoursMois) :
    print(clef, ":", valeur)
```

L'ordre d'affichage des éléments

- ne correspond pas à l'ordre de création de ces éléments
- mais il est possible d'utiliser des dictionnaires ordonnés

Structures de données - Les dictionnaires - suite

Exemple de compréhension de dictionnaire

```
lettres = "abcdefghijklmnopqrstuvwxyz"
dico = {lettres[i-1]:i for i in range(1, 27)}
for (k, v) in dico.items() :
    print(k, v)
```

Structures de données - Les dictionnaires - suite

Exemple de compréhension de dictionnaire

```
lettres = "abcdefghijklmnopqrstuvwxyz"
dico = {lettres[i-1]:i for i in range(1, 27)}
for (k, v) in dico.items() :
    print(k, v)
```

Création de dictionnaires de dictionnaires

```
if clef in dictionnaire :
    if clefSousDico in dictionnaire[clef] :
        print(dictionnaire[clef][clefSousDico])
    else : ...
else :
    dictionnaire[clef] = {}
    ...
```

Fonctions

Une fonction est un bloc d'instructions paramétré

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

print(somme(1,2))
```

Fonctions

Une fonction est un bloc d'instructions paramétré

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

print(somme(1,2))
```

Une fonction permet de factoriser du code

Fonctions

Une fonction est un bloc d'instructions paramétré

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

print(somme(1,2))
```

Une fonction permet de factoriser du code

Des fonctions particulières

- Une fonction peut ne pas renvoyer de résultat (procédure)
- Une fonction peut elle-même se rappeler (fonction récursive)

Fonctions - suite

Exemple de squelette d'une fonction récursive

```
def decompte (n) :
    if n == 0 :
        print("FIN␣!")
        return
    decompte(n-1)

decompte(10)
```

Fonctions - suite

Exemple de squelette d'une fonction récursive

```
def decompte (n) :
    if n == 0 :
        print("FIN␣!")
        return
    decompte(n-1)

decompte(10)
```

Question existentielle

Comment une variable peut-elle exister plusieurs fois en même temps ?

Fonctions - suite

Paramètres variables avec *args

```
def fonction(*args) :
    for arg in args :
        print("Bonjour", arg)
fonction("Pierre", "Paul")
```

Fonctions - suite

Paramètres variables avec *args

```
def fonction(*args) :
    for arg in args :
        print("Bonjour", arg)
fonction("Pierre", "Paul")
```

Paramètres variables sous formes de clefs/valeurs avec **kvargs

```
def fonction(**kwargs) :
    for (k, v) in kwargs.items() :
        print(k, v)

fonction(merveille1="Pyramides␣de␣Kheops",
         merveille2="Jardins␣suspendus␣...")
```

Fonctions - suite

Des fonctions peuvent être regroupées dans un module (ici monModule.py)

```
def testModule() :
    print("Je␣suis␣dans␣testModule()")
```

Fonctions - suite

Des fonctions peuvent être regroupées dans un module (ici monModule.py)

```
def testModule() :
    print("Je␣suis␣dans␣testModule()")
```

Mise en oeuvre du module monModule.py

```
#!/usr/bin/env python3

import monModule

monModule.testModule()
```

Fonctions - suite

Des fonctions peuvent être regroupées dans un module (ici monModule.py)

```
def testModule() :
    print("Je␣suis␣dans␣testModule()")
```

Mise en oeuvre du module monModule.py

```
#!/usr/bin/env python3

import monModule

monModule.testModule()
```

Le nom du module fait office d'espace de noms

Des fonctions qui n'en sont pas : les générateurs

Un exemple de générateur qui décompte à partir de 10

```
def creeGenerateur() :
    for i in range(10, 0, -1) :
        yield i

generateur = creeGenerateur()
print(next(generateur))   # affiche : 10
print(next(generateur))   # affiche : 9
```

Des fonctions qui n'en sont pas : les générateurs

Un exemple de générateur qui décompte à partir de 10

```
def creeGenerateur() :
    for i in range(10, 0, -1) :
        yield i

generateur = creeGenerateur()
print(next(generateur))   # affiche : 10
print(next(generateur))   # affiche : 9
```

Un générateur grâce au mot-clef yield:

- crée un flux de données
- qui peut être sollicité au coup par coup

Entrées/sorties - print()

La fonction print() permet d'utiliser un format

```
print("Je suis {} et j'ai {} ans", prenom, age)

print("Je suis {}, j'ai {} ans"
      .format(prenom, age))

print("Je suis %s et j'ai %2d ans"%(prenom, age))
```

Entrées/sorties - print()

La fonction print() permet d'utiliser un format

```
print("Je suis {} et j'ai {} ans", prenom, age)

print("Je suis {}, j'ai {} ans"
      .format(prenom, age))

print("Je suis %s et j'ai %2d ans"%(prenom, age))
```

Le format permet de définir des formats d'affichage ;). Exemples avec % :

- %2d
- %3.2f

Entrées/sorties - Fichiers

Lecture d'un fichier

```
fd = open("fichier", "r")
for ligne in fd :
    print(ligne, end="")
fd.close()
```

Entrées/sorties - Fichiers

Lecture d'un fichier

```
fd = open("fichier", "r")
for ligne in fd :
    print(ligne, end="")
fd.close()
```

Un fichier peut être ouvert dans différents modes. Voici les trois principaux :

- r : lecture
- w : écriture (fichier existant écrasé)
- a : écriture en ajout

Entrées/sorties - Fichiers - suite

Transfert du contenu en mémoire centrale avec readlines()

```
fd = open("fichier", "r")
for ligne in fd.readlines() :
    print(ligne, end="")
fd.close()
```

Entrées/sorties - Fichiers - suite

Transfert du contenu en mémoire centrale avec readlines()

```
fd = open("fichier", "r")
for ligne in fd.readlines() :
    print(ligne, end="")
fd.close()
```

Lecture ligne après ligne avec readline() ou next()

```
fd = open("fichier", "r")
print(fd.readline())
print(next(fd))
```

Entrées/sorties - Fichiers - suite

Transfert du contenu en mémoire centrale avec readlines()

```
fd = open("fichier", "r")
for ligne in fd.readlines() :
    print(ligne, end="")
fd.close()
```

Lecture ligne après ligne avec readline() ou next()

```
fd = open("fichier", "r")
print(fd.readline())
print(next(fd))
```

next() émet une erreur à la fin du fichier

Entrées/sorties - Fichiers - suite

Ouverture d'un fichier avec with

```
with open("fichier", "r") as fd :
    for ligne in fd :
        print(fd)
```

Entrées/sorties - Fichiers - suite

Ouverture d'un fichier avec with

```
with open("fichier", "r") as fd :
    for ligne in fd :
        print(fd)
```

Avec with le fichier est automatiquement fermé

Entrées/sorties - Fichiers - suite

Ecriture dans un fichier avec write() ou writelines()

```
fd = open("fichier", "w")
fd.write("Premiere␣ligne")
lignes = ["Seconde␣ligne", "Troisieme␣ligne"]
fd.writelines(lignes)
```

Entrées/sorties - Fichiers - suite

Ecriture dans un fichier avec write() ou writelines()

```
fd = open("fichier", "w")
fd.write("Premiere␣ligne")
lignes = ["Seconde␣ligne", "Troisieme␣ligne"]
fd.writelines(lignes)
```

Problèmes d'encodage

- Précisez l'encodage lors de l'ouverture du fichier

```
fd = open("fichier", "r", "iso-8859-15")
```

- réencoder le texte si nécessaire

```
ligneUTF = ligne.encode("utf-8")
```

Entrées/sorties - Accès à un exécutable

Accès au résultat d'un exécutable

```
import os
ll = os.popen("ls␣-l")
for ligne in ll :
    print(ligne)
```

Entrées/sorties - Accès à un exécutable

Accès au résultat d'un exécutable

```
import os
ll = os.popen("ls␣-l")
for ligne in ll :
    print(ligne)
```

os.popen()

- renvoie une liste
- et donc attention à votre mémoire...

Entrées/sorties - Accès à un exécutable

Accès au résultat d'un exécutable

```
import os
ll = os.popen("ls␣-l")
for ligne in ll :
    print(ligne)
```

os.popen()

- renvoie une liste
- et donc attention à votre mémoire...

Exécution directe d'un exécutable

```
import os
os.system("ls␣-l")
```

Entrées/sorties - Exploration d'une arborescence de dossier

Squelette type

```
import os
def parcours(repertoire) :
    print("Je␣suis␣dans␣", repertoire)
    liste = os.listdir(repertoire)
    for fichier in liste :
        if os.path.isdir(...) : ...
        if os.path.isfile(...) : ...
parcours(...)
```

Entrées/sorties - Exploration d'une arborescence de dossier

Squelette type

```
import os
def parcours(repertoire) :
    print("Je␣suis␣dans␣", repertoire)
    liste = os.listdir(repertoire)
    for fichier in liste :
        if os.path.isdir(...) : ...
        if os.path.isfile(...) : ...
parcours(...)
```

- os.listdir() : liste du répertoire
- os.path.isdir() : teste si un fichier est un dossier
- os.path.isfile() : teste si un fichier est un fichier régulier

Entrées/sorties - Gestion des exceptions

Le traitement des exceptions permet de reprendre la main en cas d'une erreur

qui sinon arrêterait l'exécution du script

Entrées/sorties - Gestion des exceptions

Le traitement des exceptions permet de reprendre la main en cas d'une erreur

qui sinon arrêterait l'exécution du script

Il repose sur trois clauses principales

- try : bloc d'instructions "protégé"
- except : bloc d'instructions à exécuter en cas d'erreurs
- else : bloc d'instructions positionné après les clauses exec

Entrées/sorties - Gestion des exceptions

Exemple de traitement des exceptions

```
try:
    fd = open("JeNExistePas", "r")
    #valeur = int("abc")

except OSError as err:
    print("OS error: {}".format(err))
    print("suite en cas d'erreur systeme")
except ValueError as err:
    print("Value Error: {}".format(err))
    print("suite sur une erreur de valeur")

else :
    print("Nb lignes =", len(fd.readlines()))
    fd.close()
```

Identification d'une sous-chaîne

```
import re
ligne = "Madame␣Claire␣Delune"
if re.search("(Monsieur|Madame)", ligne) :
    print("Ligne␣avec␣civilite")
```

Identification d'une sous-chaîne

```
import re
ligne = "Madame␣Claire␣Delune"
if re.search("(Monsieur|Madame)", ligne) :
    print("Ligne␣avec␣civilite")
```

Extractions de sous-chaînes re.search() ou re.findall()

```
l = "127.0.0.1␣-␣-␣[03/Dec/2017]␣..."
resultat = re.search("(\d+\.\d+\.\d+\.\d+)", l)
if resultat :
    print("Adresse␣IP␣:␣", resultat.group(1))
```

Bestiaire des méta-caractères

- `.` : un caractère quelconque
- `*` : de 0 à n fois
- `+` : de 1 à n fois
- `?` : de 0 à 1 fois
- `[abc]` : a ou b ou c (exemple)
- `[^abc]` : ni a, ni b, ni c (exemple)
- `"^...` : motif en début de chaîne
- `...$"` : motif en fin de chaîne
- `\` : déspécialisation d'un méta-caractère
- `(...)` : extraction
  et/ou construction d'une alternative

Exemple : extraction d'une extension

```
import re
test = "fichier.ext1.ext2"
resultat = re.search(".\.([^.]+)$", test)
# ou resultat = re.search(".+\.(.+)", test)
if resultat :
    print("Derniere␣extension␣=", resultat.group(1)
```

Exemple : extraction d'une extension

```
import re
test = "fichier.ext1.ext2"
resultat = re.search(".\.([^.]+)$", test)
# ou resultat = re.search(".+\.(.+)", test)
if resultat :
    print("Derniere␣extension␣=", resultat.group(1)
```

Exemple : extraction des extensions

```
resultats = re.findall("\.([^.]+)", test)
print(resultats)
```

Il est aussi possible d'utiliser split()

```
chaines = test.split(".")
if len(chaines) > 1 :
    print("Derniere␣extension␣=", chaines[-1])
```

Bestiaire des méta-caractères - suite

- `\d` : équivalent à `[0-9]`
- `\D` : équivalent à `[^0-9]`
- `\w` : équivalent à `[_a-zA-Z0-9]`
- `\W` : équivalent à `[^_a-zA-Z0-9]`
- `\s` : équivalent à `[ \n\r\t\f]`
- `\S` : équivalent à `[^ \n\r\t\f]`

Programmation orientée objets

Une classe définit une spécification formelle d'un concept qui réunit :

- les informations définissant un concept
- avec les traitements pouvant être effectués sur ces données (les méthodes)

Programmation orientée objets

Une classe définit une spécification formelle d'un concept qui réunit :

- les informations définissant un concept
- avec les traitements pouvant être effectués sur ces données (les méthodes)

Une classe est instanciable en un objet

L'objet est une "incarnation" du concept

Une classe implémentant le concept de Champignon

```
class Champignon :
    nomScientifique = "Fungi"

    def __init__(self, comestibilite) :
        print("Constructeur de Champignon")
        self.comestibilite = comestibilite

print(Champignon.nomScientifique)
# affiche : Fungi

amanite = Champignon("variable") # objet
print(amanite.comestibilite)
# affiche : variable
```

Interfaçage graphique

<1->

Interface graphique intégrée au script

- Tkinter : toute interface
  - est adapté de la bibliothèque créée pour le langage Tk
- Pygame : destiné au jeux
  - est basé sur la bibliothèque SDL
  - gère des sprites (matrice de pixels)

Interface graphique dans le cadre d'une architecture client-serveur

- Le script python est exécuté sur un serveur
- Son interface graphique est déportée sur le client (navigateur) : et donc utilise des technologies du web
  - HTML, CSS, SVG
  - JavaScript

Le hello world de Tkinter

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import tkinter
fenetre = tkinter.Tk()

label = tkinter.Label(fenetre, text="Hello␣World")
label.pack()

fenetre.mainloop()
```

- La classe Label permet de créer des textes
- pack() est une méthode qui intégre le label dans la fenêtre