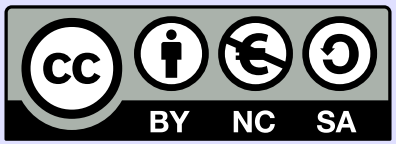
R208
Analyse et traitement de données structurées

Sami Evangelista
IUT de Villetaneuse
Département Réseaux et Télécommunications
2023–2024

http://www.lipn.univ-paris13.fr/~evangelista/cours/R208



Plan

Organisation et contenu du module

R208 — Analyse et traitement de données structurées

`http://www.lipn.univ-paris13.fr/~evangelista/cours/R208`

- Volume horaire :
  - 2×1h de cours
  - 4×3h de TP
  - 1h de contrôle
- Évaluation :
  - TP (paquet python à rendre en fin de module)
  - Contrôle
- Contenu :
  - un peu de programmation (bases de la programmation objet, fichiers) ;
  - et un peu de méthodologie de programmation (paquets, pratiques).

Plan

Les listes


```
# initialiser des listes
liste_vide = []
liste_int = [ 1, 2, 4, 8 ]
liste_bool = [ False, True, False ]
liste_mixte = [ 54, "un truc", False, 3.14 ]

# accéder à un élément d'une liste
i = liste_int[2]  # i vaut 4
b = liste_bool[3]  # exception IndexError (liste_bool a 3 éléments)

# modifier un élément d'une liste
liste_bool[2] = True  # ok => liste_bool vaut [ False, True, True ]
liste_bool[3] = True  # exception IndexError

# parcourir les éléments d'une liste
for i in liste_int: print(i)  # affiche successivement 1, 2, 4 et 8

# compter les éléments d'une liste
print("il y a", len(liste_int), "éléments dans liste_int")

# tester l'existence d'un element dans une liste
if 8 in liste_int: print("8 est bien dans list_int")

# concaténer des listes
autre_liste_int = liste_int + [ 16, 32, 64 ]
print(liste_int + [ 16, 32, 64 ])  # affiche [1, 2, 4, 8, 16, 32, 64]
```

Les dictionnaires


```
# initialiser un dictionnaire
dico = {
    "prenom": "tom",
    "nom": "pouce",
    "age": 22
}

# récupérer la valeur associée à une clé
nom = dico["nom"]
adresse = dico["adresse"]  # exception KeyError

# modifier la valeur associée à une clé
dico["age"] = 24
dico["adresse"] = "Angleterre"

# parcourir les clés d'un dictionnaire
for cle in dico:
    print(cle, dico[cle])

# compter le nombre de clés dans un dictionnaire
print("il y a", len(dico), "éléments dans liste_int")

# tester qu'une clé est bien présente dans un dictionnaire
if "adresse" in dico:
    print("l'adresse est connue")
```

Les fonctions

```
# une fonction discriminant avec trois paramètres : a, b, c
def discriminant(a, b, c):
    return (b ** 2) - (4 * a * c)

x = discriminant(5, 10, 5)  # appel de la fonction
print(x)  # x vaut 10 ** 2 - 4 * 5 * 5 = 0

# une fonction mult avec deux paramètres ayant une valeur par défaut
def mult(a, b=2, c=1):
    return a * b + c

print(mult(4, 5, 2))  # affiche 4 * 5 + 2 = 22
print(mult(4))  # affiche 4 * 2 + 1 = 9
print(mult(4, c=10))  # affiche 4 * 2 + 10 = 18
```

Les exceptions


```
try:  # début d'un bloc d'instructions pouvant lever des exceptions
    x = int(input("Combien vaut x ? "))  # peut lever ValueError
    y = int(input("Combien vaut y ? "))  # peut lever ValueError
    print("x / y vaut", x / y)  # peut lever ZeroDivisionError
except ValueError:  # si ValueError levée
    print("La valeur saisie n'est pas un nombre!")
except ZeroDivisionError:  # si ZeroDivisionError levée
    print("y vaut zéro!")
except:  #  tout autre type d'exception
    print("Une erreur s'est produite!")

#  fonction qui peut lever une exception ValueError
def factorielle(n):
    if n < 0:
        raise ValueError
    resultat = 1
    for i in range(n):
        resultat = resultat * (i + 1)
    return resultat

try:
    print("5! vaut", factorielle(5))
    print("-2! vaut", factorielle(-2))
except ValueError:
    print("un factoriel ne peut pas être calculé!")
```

Plan

Lisibilité du code

- La lisibilité du code est un aspect fondamental.
- Un code pénible à lire sera difficile à comprendre, donc à corriger (débuguer) et à faire évoluer.
- Quelques bonnes pratiques :
  - respecter l'indentation (remarque inutile avec python mais importante dans le cas général)
  - utiliser des noms de variables explicites
  - commenter le code
  - bouts de code pas trop gros :
    - lignes $\leq$ 80–120 caractères
    - fonctions $\leq$ 30–50 lignes (au-delà $\Rightarrow$ on divise en sous-fonctions)
    - fichier $\leq$ 1 000 lignes (au-delà $\Rightarrow$ on sépare en modules)
  - utiliser des docstrings
- Il existe des outils d'analyse de code qui permettent de vérifier le respect de règles de codage.

  pour python : voir pylint

Les docstring python

- docstring = chaîne de caractères placée en début de fonction, ou de module ou de classe et donnant une description succinte de son utilité
- Convention : utiliser trois double quotes (""") pour les docstrings.
- Une docstring n'est pas un commentaire : elle n'est pas ignorée par python mais sert de message d'aide (fonction `help` appliquée sur l'élément).
- Exemple :

Module calcul :

```
"""Fournit des fonctions de calcul."""

def somme(x, y):
    """Renvoie la somme des deux entiers x et y."""
    return x + y
```

```
>>> import calcul
>>> help(calcul)
Help on module calcul:
NAME
    calcul - Fournit des fonctions de calcul.
FUNCTIONS
    somme(x, y)
        Renvoie la somme des deux entiers x et y.
```

- Quand un programme devient long on a intérêt à le diviser en **modules**.
- Un module, c'est un regroupement logique de composants réutilisables (des fonctions, des variables, des classes, ...).
- Avantages :
  - code plus facile à maintenir (corriger les bugs) et à faire évoluer
  - permet la réutilisabilité ($\Rightarrow$ les modules peuvent être facilement réutilisés par d'autres programmes)
- Concrètement, en python, un module est un fichier d'extension `.py`.
- On peut ensuite l'importer dans sa totalité (`import le_module`) ou en partie (`from le_module import un_element`).
- Lors du premier import toutes les instructions se trouvant dans le module sont exécutées.

Exemple d'import de module

Module `calcul` dans le fichier `calcul.py` :

```
"""Module avec des fonctions de calcul."""

print("Bienvenue dans le module de calcul.")

def produit(a, b):
    return a * b

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

Utilisation du module `calcul` :

```
>>> import calcul
Bienvenue dans le module de calcul.
>>> import calcul  #  module déjà importé (=> print non exécuté)
>>> print(calcul.produit(3, 4))  # notation pointée nécessaire
12
>>> from calcul import somme
>>> print(somme(2, 5))  # pas besoin de faire calcul.somme
7
```

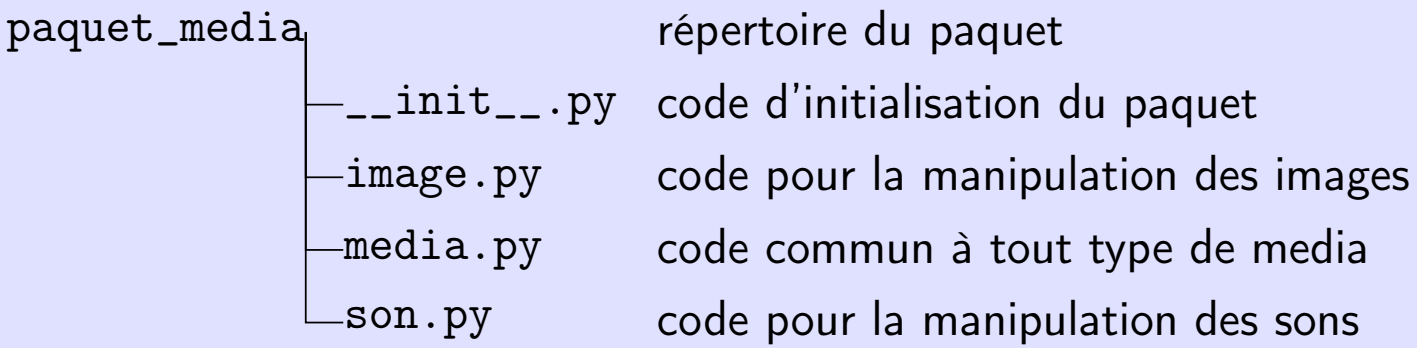
Les paquets

- Python fournit un niveau de structuration supplémentaire : **les paquets**.
- Un paquet est un regroupement logique de modules ou de sous-paquets dans un répertoire.
- Pour qu'un répertoire soit considéré comme un paquet par python, il faut qu'il contienne un fichier `__init__.py` (même vide).
- Ce fichier contient le code d'initialisation du paquet, exécuté une seule fois lors du premier import par python du paquet ou d'un de ses composants.
- Exemples de code d'initialisation :
  - afficher le numéro de version du paquet
  - initialiser des variables globales
  - créer des fichiers nécessaires au fonctionnement du paquet
- Les paquets sont importés comme les modules.
- On peut dans les modules d'un paquet, faire des **imports relatifs** qui permettent d'importer un module du même paquet :

```
from . import un_autre_module_du_meme_paquet
from .un_autre_module_du_meme_paquet import un_truc
```

Les paquets — Exemple

- Soit un paquet permettant de gérer des fichiers media (sons, images, ...) :



fichier `media.py` :

```
var = "blabla"

def ouvre_media(fichier):
    print("blabla")
```

fichier `son.py` :

```
# importe ouvre_media du module
# media du même paquet
from .media import ouvre_media
ouvre_media("...")
```

fichier `image.py` :

```
# importe le module media
# du même paquet
from . import media
print(media.var)
media.ouvre_media("...")
```

Hors du paquet, pour importer :

```
from paquet_media import son
from paquet_media.media \
   import ouvre_media

ouvre_media("...")
```

Plan

Objets et classes

- La programmation orientée objet (POO) est une méthode de structuration des programmes qui vise à représenter les données manipulées par un programme comme des entités appelées **objets**.
- Un objet est une boîte contenant
  - des **attributs** (ou propriétés) : les variables que l'on trouve dans l'objet ;
  - et des **méthodes** : les opérations que l'on peut réaliser sur l'objet.
- Des objets possédant les mêmes caractéristiques (attributs et méthodes) forment une **classe**.

  On dit que les objets sont les **instances** de la classe.
- Exemple :
  - On a une classe `Animal`.

    Convention : première lettre des noms de classes en majuscule.
  - Les attributs des objets instances de cette classe sont :
    - `nom`
    - `espece`
    - `age`
  - et les méthodes que l'on peut leur appliquer sont :
    - `afficher` — affiche les attributs de l'animal sur la sortie standard
    - `ajoute_annees` — ajoute des années à l'âge de l'animal

Les attributs

- attribut = variable stockée dans un objet
- Pour manipuler un attribut `a` d'un objet `o`, on utilise la notation pointée :

`o.a`

- Par exemple :

```
point.x = 12
point.y = 14
point.x = point.x + 20
print("coordonnées du points = (", point.x, ",", point.y, ")")
```

Les méthodes

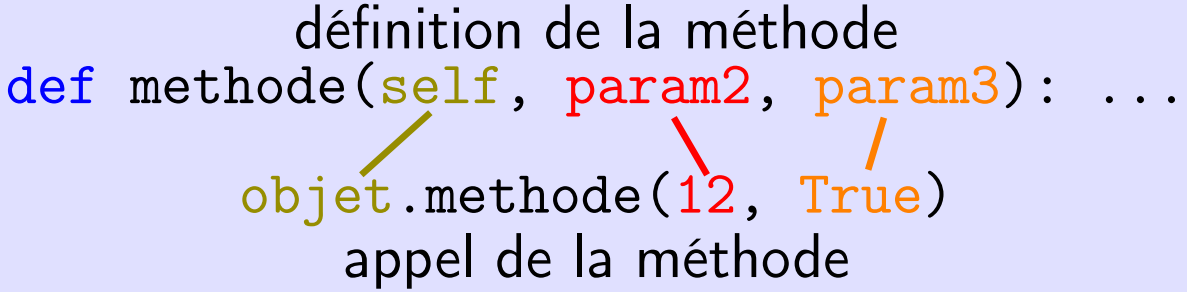
- méthode = opération que l'on peut réaliser sur un objet
- Une méthode se déclare comme une fonction à l'intérieur de sa classe.
  - On l'indente donc toujours d'un niveau.
- Elle a toujours pour premier paramètre l'objet `self` qui est l'objet sur lequel on appelle la méthode.

  $\Rightarrow$ `self` est donc toujours une instance de la classe dans laquelle est déclarée la méthode.
- À l'intérieur de la méthode on peut manipuler les attributs de `self` avec la notation pointée.
  - Par exemple : `self.nom = "Baloo"`, `print(self.age)`
- Les attributs de `self` peuvent être modifiés dans la méthode et ses modifications sont persistantes (i.e., visibles après l'exécution de la méthode).
- Exemple de classe avec deux méthodes :

```
class Animal:
    def afficher(self):  # self est une instance de la class Animal
        print(self.nom, "est un", self.espece, "de", self.age, "ans")
    def ajoute_annees(self, annees):
        self.age = self.age + annees
```

Appeler une méthode

- On a vu qu'une méthode = fonction dans une classe `C` dont le premier paramètre `self` est une instance de `C`.
- On appelle une méthode un peu comme une fonction sauf que l'on sort le premier argument et on utilise la notation pointée :



- Exemple :

```
class Animal:
    def afficher(self):
        print(self.nom, "est un", self.espece, "de", self.age, "ans")
    def ajoute_annees(self, annees):
        self.age = self.age + annees

a = Animal("gros-minet", "chat", 6)  # initialise l'objet (cf. suite)
a.ajoute_annees(10)  # appel de ajoute_annees <=> a.age = a.age + 10
a.afficher() # affiche "gros-minet est un chat de 16 ans"
```

Création et initialisation d'un objet

- On utilise la méthode spéciale `__init__` pour créer un objet et initialiser ses attributs.
- Pour appeler la méthode `__init__(self, param1, ..., paramN)` d'une classe `C` :

  `C(arg1, ..., argN)`

  (`self` n'apparaît pas dans l'appel)
- Exemple :

```
class Animal:
    def __init__(self, nom, espece, age):
        self.nom = nom
        self.espece = espece
        self.age = age

gros_minet = Animal("gros-minet", "chat", 6)  # appel de __init__
print(gros_minet.age)      # affiche "6"
print(gros_minet.espece)   # affiche "chat"
```

Différence entre objet et référence

- Le code suivant :

```
a = Animal("gros-minet", "chat", 6)
b = a
b.ajoute_annees(4)
a.afficher()
```

  affichera *gros-minet est un chat de 10 ans*.

- Raison :
  - L'instruction `b = a` ne crée pas un nouvel objet en mémoire.
  - Elle définit un nouveau symbole `b` qui **référence** l'objet déjà référencé par `a`.

  $\Rightarrow$ On a donc un seul objet en mémoire accessible via 2 références (`a` et `b`).

  $\Rightarrow$ `a.ajoute_annees(4)` et `b.ajoute_annees(4)` sont équivalents.

- Pour s'en convaincre on peut utiliser la fonction `id` qui affiche l'identité d'un objet ($\approx$ son adresse en mémoire) :

```
a = Animal("gros-minet", "chat", 4)
b = a
c = Animal("gros-minet", "chat", 4)
print(id(a) == id(b))  # affiche True
print(id(a) == id(c))  # affiche False
```

Des objets partout

- En python, tout est objet : les entiers (classe `int`), les chaînes de caractères (classe `str`), les booléens (classe `bool`), les réels (classe `float`), les listes (classe `list`), les dictionnaires (class `dict`), les modules (classe `module`), ...
- Quelques méthodes de la classe `list` :
  - `l.append(x)` ajoute `x` en fin
  - `l.insert(i, x)` ajoute `x` en position `i`
  - `l.pop(i)` retire l'élément en position `i` (`IndexError` si `i` $\notin$ [0..`len(l)-1`])
  - `l.remove(x)` retire la première occurence de `x`
- Quelques méthodes de la classe `dict` :
  - `d.get(c)` renvoie la valeur associée à la clé `c` ou `None` si `c` n'est pas dans `d`
  - `d.pop(c)` retire la clé `c` et renvoie la valeur qui lui était associée

Les méthodes cachées

- La méthode `__init__` est une méthode cachée :
  - Le programmeur qui utilise la classe ne l'appellera jamais explicitement.
  - C'est l'interpréteur python qui l'appellera à la création d'un objet.
- Il y a de nombreuses autres méthodes cachées qui seront utilisées par python dans des contextes particuliers, par exemple :
  - `__lt__` : pour pouvoir comparer des objets de la classe (lt = less than)
  - `__str__` : pour pouvoir convertir l'objet en chaînes de caractère
- Exemple :

```
class Animal:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age
    def __str__(self):
        return self.nom + " (" + str(self.age) + " ans)"
    def __lt__(self, autre):
        return self.age < autre.age

scooby = Animal("scooby-doo", 7)
print(scooby)  # appel de __str__ (=> affiche "scooby-doo (7 ans)")
sonic = Animal("sonic", 4)
if sonic < scooby:  # appel de __lt__(avec self=sonic, autre=scooby)
    print("sonic est plus jeune que scooby-doo")
```

Plan

Manipulation des fichiers

(on s'intéresse uniquement aux fichiers texte)

- Manipulation typique d'un fichier :
  1. ouverture (fonction `open` paramétrée par le chemin du fichier et un **mode**)
  2. écriture ou lecture de données
  3. fermeture (méthode `close`)
- Modes d'ouverture d'un fichier :
  - `'r'` ⇒ ouverture en lecture (mode par défaut)
    - Le fichier doit exister. Sinon ⇒ exception.
  - `'w'` ⇒ ouverture en écriture
    - Si le fichier existe il est écrasé. Sinon il est créé.
  - `'a'` ⇒ ouverture en ajout (⇔ écriture en fin de fichier)
    - Si le fichier n'existe pas il est créé.
- Pour lire dans un fichier `f` (ouvert avec mode=`'r'`) :
  - `f.read` pour lire tout le contenu
  - `f.readlines` pour lire le contenu ligne par ligne (renvoie la liste des lignes)
- Pour écrire dans un fichier `f` (ouvert avec mode=`'w'` ou mode=`'a'`) :
  - `f.write` écrit une chaîne de caractères

Exemple de manipulation de fichiers


```
f = open("bonjour.txt", "w")  # mode "w" => ouverture en écriture
f.write("salut tout\nle monde")  # \n = saut de ligne
f.close()


f = open("bonjour.txt")  # mode non précisé => ouverture en lecture
contenu = f.read()
print(contenu)  # affiche
                # salut tout
                # le monde
f.close()


f = open("bonjour.txt", "a")  # mode "a" => ouverture en mode ajout
f.write("\nca va bien?")  #  écrit une nouvelle ligne "ca va bien?"
                         #  à la suite de "le monde"
f.close()
```

Exceptions sur les fichiers

Manipuler des fichiers peut causer la levée d'exceptions, principalement :

- `FileNotFoundError` — fichier inexistant (sur `open` avec mode='r')
- `PermissionError` — erreur de droit
- `IsADirectoryError` — tentative d'ouverture d'un répertoire

```
# Exemple : un programme qui lit et affiche le contenu d'un fichier
nom_fichier = input("Entrez le chemin du fichier à lire\n")
try:
    f = open(nom_fichier)
    print(f.read())
    f.close()
except FileNotFoundError:
    print("Ce fichier n'existe pas!")
except PermissionError:
    print("Je n'ai pas le droit de lire ce fichier!")
except IsADirectoryError:
    print("C'est un répertoire!")
```

Stockage des données

Où stocker les données manipulées par un programme ?

- Dans une base de données type relationnel
  - \+ sécurité et intégrité des données
  - \+ rapidité de traitement des requêtes
  - \+ stockage efficace
  - \+ permet des accès concurrents (p.ex., modifications simultanées)
  - – fichiers binaires
  - – lourd à mettre en œuvre (nécessite un SGBD, utilisation de requêtes)
- Dans des fichiers structurés (p.ex., XML, CSV, JSON)
  - \+ fichiers textes ($\Rightarrow$ lisibles et modifiables par un humain)
  - \+ facilement manipulables par des scripts
  - \+ adapté pour les échanges de données entre applications
  - – difficile de mettre en œuvre des accès concurrents
  - – lent pour des gros volumes de données

- JSON = JavaScript Object Notation
  (format dérivé du langage javascript)
- décrit dans la RFC 8259
- dernière version : déc. 2017
- très utilisé comme format d'échange de données par les services web

Le format JSON — Les types

Valeurs JSON valides :

- la valeur `null` (équivalent du `None` en python)
- des nombres entiers (p.ex., `-1`, `128`) ou réels (p.ex., `3.14`)
- des booleéns (`true`, `false`)
- des chaînes de caractères
- des listes de valeurs JSON
- des dictionnaires de valeurs JSON

Quelques restrictions :

- Impossible de coder des tuples.
- Impossible d'insérer des commentaires dans les fichiers JSON.
- Les chaînes de caractères doivent être écrites sur une seule ligne.
- Les clés des dictionnaires sont nécessairement des chaînes de caractères.

Le format JSON — Un exemple


```
[
  {
    "titre": "Doctor Strange in the Multiverse of Madness",
    "date_sortie": "04/05/2022",
    "notes": 3.1,
    "vu": false,
    "avis": [
      "bof", "pas terrible"
    ]
  },
  {
    "titre": "Les Animaux Fantastiques : les Secrets de Dumbledore",
    "date_sortie": "13/04/2022",
    "notes": null,
    "vu": true,
    "avis": []
  }
]
```

Le module json

- Le module `json` fournit des fonctions permettant de traduire des données au format JSON et inversement.
- 2 fonctions principales :
  - `dumps` convertit un objet en une chaîne de caractères JSON
  - `loads` convertit une chaîne de caractères JSON en un objet
- Exemple :

```
import json

# exemple 1: écriture d'un dictionnaire dans un fichier
dico = {"a": True, "b": [1, 2, 3]}
f = open("fichier.txt", "w")
f.write(json.dumps(dico))
f.close()

# exemple 2: récupération du dictionnaire depuis le fichier
f = open("fichier.txt")
d = json.loads(f.read())
print(d["a"])  # affiche True
print(d["b"])  # affiche [1, 2, 3]
f.close()
```

- Attention, `dumps` prend en paramètre une valeur JSON valide (diapo. 31).
- Si `dumps` rencontre une valeur invalide ⇒ exception `TypeError`.
- Ce sera le cas si on lui passe en argument un objet quelconque.
- On doit alors convertir cet objet dans une valeur JSON valide, par exemple en définissant une méthode réalisant cette conversion.
- Exemple :

```
import json

class Animal:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age
    def convertit_en_json(self):
        return {
            "nom": self.nom,
            "age": self.age
        }

scooby = Animal("scooby-doo", 7)
pjson = json.dumps(scooby)  # TypeError
pjson = json.dumps(scooby.convertit_en_json())  # OK
```