Python, administration système

Hugo Mougard

https://www.orsys.fr/

Table des matières

Informations sur la formation

Présentation & coordonnées

Nom	Hugo Mougard
Courriel	hugo@mougard.fr
Activité	Formation & consulting en Machine Learning, Deep Learning & Python
Spécialité	Traitement du langage naturel et du code

Objectifs

- Maîtriser les bases du langage Python
- Posséder une vue d'ensemble des librairies Python disponibles pour les tâches d'administration système
- Être capable d'installer une librairie Python
- Savoir réaliser des tâches d'administration système avec des scripts Python
- Comprendre la syntaxe objet de Python

Prérequis

- Avoir suivi le stage *Initiation à la programmation avec Python (THO)* ou posséder des connaissances de base en algorithmique (variables, tableaux, fonctions)
- Avoir un **compte Google** afin de pouvoir faire les TPs dans Google Colaboratory

Programme

- Les bases du langage Python
- Un peu plus loin avec Python
- Les bases de l'administration système
- Compléments d'administration système

Ressources

Les supports utilisés vous seront remis à chaque début de cours.

Emploi du temps

- 9h à 12h30 et de 14h à 17h30

Tour de table : présentez-vous !

- Votre nom
- Votre métier
- Votre société client si applicable
- Vos compétences dans les domaines liés à cette formation
- Vos objectifs et vos attentes vis-à-vis de cette formation

Introduction au langage Python

Historique

1989 Création du langage par Guido Van Russum

2001 Lancement de la Python Software Foundation

2001 Passage en GPL

2009 Python 3

Logo Python, Python Software Foundation (”PSF”), PSF trademark.

Caractéristiques

Python est :

Interprété Et compilé à la volée, modules en C

Orienté objet (mais pas que)

Portable Compatible avec toutes les plateformes actuelles

Flexible Couteau suisse, de l'admin système au webdev

Populaire Top 5 des langages les plus utilisés depuis des années

Points forts/faibles

Atouts

- Stable
- Multi-plateforme
- Facile à apprendre
- Grande communauté (le plus utilisé depuis 2019)
- « Batteries included » : un besoin, un module

Inconvénients

- Non-compilé
  - Plus lent qu'un langage bas-niveau
  - Optimiser une opération ⇒ pas facile à apprendre

Plates-formes

Différents interpréteurs :

- CPython/Pypy ⇒ C/C++
- Jython ⇒ JVM
- IronPython ⇒ .Net

Domaines

Domaines d'applications :

- Web (Django ,Flask, ...)
- Sciences (Data mining, Machine learning, Physique, ...)
- OS (Linux, Raspberry, Script administration système, ...)
- Éducation (Initiation à la programmation)
- CAO 3D (FreeCAD, pythonCAD, ...)
- Multimédia (Kodi, ...)

Environnements de travail

Les trois principaux IDEs Python sont :

Visual Studio Code IDE gratuit proposé par Microsoft

PyCharm IDE en double version gratuite/professionnelle proposé par JetBrains

Spyder IDE open source et gratuit orienté data science

Ils intègrent tous les outils modernes de développement.

Cependant vous pouvez bien sûr utiliser votre éditeur de texte favori (emacs, vim, atom, sublime text, ...).

Versions 2 et 3 de Python

Version 2 Plus supportée depuis le premier janvier 2020. Toujours présente dans les bases legacy.

Version 3 Version actuelle de Python. Tous les nouveaux développements doivent l'utiliser.

Ressources

- Documentation officielle Python
- Stack Overflow
- Reddit (forum) d'apprentissage de Python
- Python Tutor

Installation

Consulter le très bon billet de blog Real Python sur le sujet.

Avez-vous des questions ?

Bases du langage Python

Bases du langage Python

Support de cours

Ressource complémentaire

Lien vers le support de cours complet

Bases du langage Python

Types de base et variables

Introduction aux variables

Une variable est un *label* assigné à un objet Python.

On peut utiliser ce label à n'importe quel moment en lieu et place de l'objet lui-même.

Création d'une variable

Pour créer une variable, on utilise la forme suivante :

```
answer = 42
```

Utilisation d'une variable

Pour utiliser une variable, on utilise son nom :

```
>>> answer = 42
>>> print(answer)
42
```

Nommage

On peut nommer ses variables en respectant ces deux règles :

- Le nom commence par une lettre minuscule ou majuscule ou un tiret bas
- Le nom continue par ces mêmes caractères ou des chiffres

Unicode

Depuis Python 3, on peut utiliser de l'unicode. *C'est toutefois très déconseillé*

Types de base

Pour représenter des valeurs simples, on utilise principalement :

`int`	Nombres entiers
`float`	Nombres réels
`str`	Chaînes de caractères
`bool`	Booléens

La fonction `type`

Permet de connaître le type d'un objet :

```
>>> question = "Quelle est la réponse à l'univers ?"
>>> answer = 42
>>> type(question)
<class 'str'>
>>> type(answer)
<class 'int'>
```

Conversion de types

On peut utiliser les classes représentants les types pour convertir vers ceux-ci :

```
>>> float(42)
42.0
```

Avez-vous des questions ?

Travaux pratiques

Instructions

Bases du langage Python

Affichage

Fonction d'affichage

`print` est la principale fonction pour afficher du texte en Python :

```
>>> print("Hello World!")
Hello World!
```

Arguments de `print`

`sep` Affiché entre chaque chaîne argument. Défaut : un espace

`end` Affiché en fin d'affichage. Défaut : un retour chariot

`file` Fichier de sortie. Défaut : **`sys.stdout`**

`flush` Faut-il flush le buffer ? Défaut : **`False`**

```
>>> print("Hello", "World", sep=", ", end="!\n")
Hello, World!
```

Chaînes formatées

Il est possible de construire des chaînes complexes avec la méthode `format` :

```
>>> "My name is {nom}, {prenom} {nom}.".format(prenom="James",
...                                            nom="Bond")
'My name is Bond, James Bond.'
```

Le site PyFormat donne des précisions sur toutes les options disponibles.

f-strings

On peut aussi utiliser un raccourci syntaxique pour appeler `format` :

```
>>> prenom = "James"
>>> nom = "Bond"
>>> f"My name is {nom}, {prenom} {nom}."
'My name is Bond, James Bond.'
```

Avez-vous des questions ?

Travaux pratiques

Instructions

Bases du langage Python

Collections

Introduction

Les quatre principales structures de données en Python sont :

`list`	Listes modifiables et potentiellement hétérogènes
`dict`	Associations clef-valeur
`tuple`	N-uplets non modifiables
`set`	Ensembles (pas de répétition, pas d'ordre)

Chacune a ses caractéristiques propres.

Caractéristiques de `list`

- Collection la plus utilisée en Python
- Peut contenir des éléments de types différents (*très déconseillé*)
- A de bonnes performances pour un usage standard
- Plus proche d'un tableau que d'une liste chaînée (accès $\mathcal{O}(1)$ par exemple)
- Pas adaptée au calcul scientifique (pour ça, utiliser `numpy`)

Manipulation basique d'une liste

```
>>> l = []                  # Création d'une liste vide
>>> l = [1, 2, 3]           # Création avec éléments
>>> l[0]                    # Accès par indiçage positif
1
>>> l[1]
2
>>> l[-1]                   # Accès par indiçage négatif
3
>>> l[-2]
2
>>> l.append(4)             # Ajout d'un élément
>>> l.extend([5, 6])        # Ajout de plusieurs éléments
>>> l.sort(reverse=True)    # Tri de liste, ici inversé
>>> len(l)                  # Taille de la liste
6
>>> del l[4]                # Suppression par indice
>>> l.remove(1)             # Suppression par valeur
>>> l
[6, 5, 4, 3]
```

Tranche de liste

Une tranche de liste sélectionne une sous-partie de la liste :

```
>>> l = [0, 1, 2, 3, 4, 5]
>>> l[0:3]
[0, 1, 2]
>>> l[:3]
[0, 1, 2]
>>> l[3:]
[3, 4, 5]
>>> l[0:3:2]
[0, 2]
```

Caractéristiques de `dict`

`dict` permet de définir des associations clef-valeur :

- Correspond aux objets JS, Map Java
- Accès à la valeur associée à une clef en $\mathcal{O}(1)$
- Test de présence d'une clef en $\mathcal{O}(1)$

Utilisation d'un `dict`

```
>>> d = {}                                  # Création d'un dictionnaire vide
>>> d = {"a": 1, "b": 2}                    # Création avec éléments initiaux
>>> d = dict([("a", 1), ("b", 2)])          # Création à partir d'un itérable
>>> d = dict(a=1, b=2)                      # Création par arguments mots-clefs
>>> d["a"]                                  # Accès à une valeur
1
>>> d["c"] = 3                              # Modification ou ajout d'une valeur
>>> "a" in d                                # Test d'appartenance
True
>>> "d" in d
False
>>> 1 in d
False
>>> len(d)                                  # Taille d'un dictionnaire
3
>>> del d["a"]                              # Suppression d'un élément
>>> d
{"b": 2, "c": 3}
```

Caractéristiques de `tuple`

Les n-uplets (*tuples* en anglais), sont très similaires aux listes, mais sont :

- Immuables
- Adaptés aux éléments hétérogènes
- Hashables si leurs éléments sont immuables
- Très utiles pour retourner plusieurs éléments dans une fonction

Utilisation d'un `tuple`

```
>>> t = ()          # Création d'un n-uplet vide
>>> t = (1, 2, 3)   # Création avec éléments
>>> t[0]            # Accès à un élément par indiçage positif
1
>>> t[1]
2
>>> t[-1]           # Accès à un élément par indiçage positif
3
>>> t[-2]
2
>>> 1 in t          # Test d'appartenance
True
>>> 4 in t
False
>>> t + (4, 5, 6)   # Concaténation
(1, 2, 3, 4, 5, 6)
>>> len(t)          # Taille du n-uplet
3
```

Définitions multiples avec un `tuple`

```
>>> a, b = 0, 1
>>> a
0
>>> b
1
```

Retours multiples avec un `tuple`

```
>>> def sum_and_product(a, b):
...     return a + b, a * b
...
>>> x, y = sum_and_product(3, 4)
>>> print(f"Somme : {x}, produit : {y}")
Somme : 7, produit : 12
```

Échange de variables avec un `tuple`

```
>>> a = 1
>>> b = 2
>>> a, b = b, a
>>> a
2
>>> b
1
```

Utilisé pour implémenter le tri à bulles :

```
def bubble_sort(arr):
    for i in range(1, len(arr)):
        while i > 0 and arr[i] < arr[i - 1]:
            arr[i], arr[i - 1] = arr[i - 1], arr[i]
            i -= 1
```

Caractéristiques de `set`

`set` définit des collections non ordonnées sans répétition :

- Accès aux opérations d'ensembles (union, intersection, …)
- Test de présence d'un objet en $\mathcal{O}(1)$ (contre $\mathcal{O}(n)$ pour une `list`)

```
>>> import timeit
>>> timeit.timeit(setup="l = list(range(1_000_000))",
...               stmt="1_000_000 in l",
...               number=1_000)
6.942514133999794
>>> timeit.timeit(setup="s = set(range(1_000_000))",
...               stmt="1_000_000 in s",
...               number=1_000)
2.4280000616272446e-05
```

Utilisation d'un set

```
>>> s = set()                          # Création d'un ensemble vide
>>> s = {"a"}                          # Création avec éléments
>>> s = set(["a"])                     # Création à partir d'un itérable
>>> s.add("b")                         # Ajout d'un élément
>>> s.update(["c", "d"])               # Ajout de plusieurs éléments
>>> s
{'c', 'b', 'd', 'a'}
>>> "a" in s                           # Test d'appartenance
True
>>> "e" in s
False
>>> s | {"e", "f"}                     # Union
{'c', 'b', 'e', 'a', 'f', 'd'}
>>> s & {"a", "e"}                     # Intersection
{'a'}
>>> s - {"a", "e"}                     # Différence
{'d', 'c', 'b'}
>>> s < {"a", "b", "c", "d", "e"}      # Test de sous-ensemble
True
```

`sets` immuables

L'alternative `frozenset` est disponible pour des sets immuables.

Équivalent de `tuple` vis à vis de `list`.

Avez-vous des questions ?

Travaux pratiques

Instructions

Bases du langage Python

Structures de contrôle

Introduction

Python utilise principalement trois structures de contrôle de flot : `if`, `while` et `for`.

Donc pas de `switch` comme dans d'autres langages.

Nous verrons plus tard qu'il existe aussi une structure `try`/`except` pour la gestion d'erreurs.

Il existe aussi une instruction `match` depuis Python 3.10 mais non traitée ici car trop compliquée.

Structure conditionnelle

Utilisation de `if`, `elif` et `else` :

```
>>> age = 23
>>> if age < 18:
...     print("Mineur en France")
... elif age < 21:
...     print("Mineur aux États-Unis")
... else:
...     print("Majeur")
...
Majeur
```

Structure de boucle « Pour dans » · Avec des indices

La fonction `range` permet de créer des suites d'indices :

```
range(10)              # 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
range(1, 10)           # 1, 2, 3, 4, 5, 6, 7, 8, 9
range(1, 10, 2)        # 1, 3, 5, 7, 9
list(range(1, 10, 2))  # [1, 3, 5, 7, 9]
```

Par défaut elle renvoie des objets de type `range`, plus efficaces que des listes.

Utilisation dans une boucle :

```
>>> for i in range(2):
...     print(i)
...
0
1
```

Structure de boucle « Pour dans » · Avec une liste

```
>>> l = ["a", "b"]

>>> for item in l:
...     print(item)
...
a
b

>>> for i in range(len(l)):
...     print(i, l[i])
...
0 a
1 b

>>> for i, item in enumerate(l):
...     print(i, item)
0 a
1 b
```

Structure de boucle « Pour dans » · Avec un dictionnaire

```
>>> d = {"a": 1, "b": 2}

>>> for key in d:
...     print(key)
...
a
b

>>> for key, value in d.items():
...     print(key, value)
...
a 1
b 2
```

Structure de boucle « Pour dans » · Avec plusieurs itérables

Utilisation de la fonction `zip` :

```
>>> l1 = ["a", "b", "c"]
>>> l2 = [True, False]

>>> for item1, item2 in zip(l1, l2):
...     print(item1, item2)
...
a True
b False

>>> import itertools
>>> for item1, item2 in itertools.zip_longest(l1, l2):
...     print(item1, item2)
...
a True
b False
c None
```

Fonctionne avec un nombre arbitraire d'itérables.

Structure de boucle « tant que »

```
>>> ok = False
>>> while not ok:
...     answer = input("Entrez o ou n: ")
...     ok = answer in {"o", "n"}
...
Entrez o ou n: oui
Entrez o ou n: o
>>> answer
'o'
```

break & continue

break Utilisé pour sortir de la boucle la plus intérieure

continue Utilisé pour passer directement à l'itération suivante de la boucle la plus intérieure

```
>>> i = 0
>>> while True:
...     i += 1
...     if i < 10:
...         continue
...     if i > 11:
...         break
...     print(i)
...
10
11
```

Clause `else` pour les boucles

La clause **`else`** est exécutée si on sort de la boucle sans **`break`** :

```
>>> for i in range(1, 4):
...     print(i)
... else:
...     print("Soleil")
...
1
2
3
Soleil

>>> for i in range(1, 4):
...     if i == 3:
...         break
...     print(i)
... else:
...     print("Soleil")
...
1
2
```

Avez-vous des questions ?

Travaux pratiques

1. Instructions
2. Instructions

Bases du langage Python

Lecture et écriture de fichiers

Mécanismes principaux

- Fonction `open` pour récupérer un objet pour manipuler le fichier
- Utilisation d'un bloc `with` pour garantir la fermeture du fichier

Bloc `with`

Définit un contexte :

- L'entrée et la sortie du contexte sont gérés par le gestionnaire de contexte
- Les erreurs peuvent aussi l'être

Très utile pour la gestion de fichier : **open** est un gestionnaire de contexte.

Paramètres les plus importants d'open

`mode` Par défaut, `"r"` pour lecture. Voir ici pour la liste complète.

`encoding` Par défaut, dépend de la locale. Quasi toujours mettre `encoding="utf8"`.

Lecture

```
>>> with open("example.txt", encoding="utf8") as fh:
...     print(fh.read())
...
Ma première ligne
Ma deuxième ligne

>>> with open("example.txt", encoding="utf8") as fh:
...     for line in fh:
...         print(line)
...
Ma première ligne

Ma deuxième ligne


>>> with open("example.txt", encoding="utf8") as fh:
...     lines = fh.readlines()
...     print(len(lines))
...
2
```

Écriture

```
with open("example.txt", mode="w", encoding="utf8") as fh:
    fh.write("Mon texte\n")

with open("example.txt", mode="w", encoding="utf8") as fh:
    fh.writelines(["Ma première ligne\n", "Ma deuxième ligne\n"])
```

Manipulation de deux fichiers à la fois

```
with open("example.txt", encoding="utf8") as fh_read, open(
    "example-upper.txt", mode="w", encoding="utf8"
) as fh_write:
    fh_write.write(fh_read.read().upper())
```

Sauts de ligne

Par défaut, Python 3 comprend en lecture les sauts de ligne de Windows, Mac OS et GNU/Linux sans travail additionnel.

En écriture il utilise `\n` (sauts de ligne GNU/Linux).

Avez-vous des questions ?

Travaux pratiques

Instructions

Bases du langage Python

Fonctions

Introduction

Les fonctions sont un des deux mécanismes (avec les classes) pour regrouper du code.

Elles sont définies avec `def` suivi de leur nom, une liste d'arguments entre parenthèses, `:` puis le corps de la fonction.

On utilise le mot-clef `return` pour renvoyer une valeur.

Exemple

```
>>> def add(a, b):
...     return a + b
...
>>> add(1, 2)
3
```

Valeurs par défaut

On peut spécifier des valeurs par défaut aux arguments :

```
>>> def add(a, b=1):
...     return a + b
...
>>> add(1, 2)
3
>>> add(1)
2
```

Valeurs par défaut mutables

Attention

Ne jamais donner de valeur par défaut mutable à une fonction.

```
>>> def append_to(item, lst=[]):
...     lst.append(item)
...     return lst
...
>>> append_to(1)
[1]
>>> append_to(1)
[1, 1]
```

Appel par mot-clef

On peut spécifier le nom de la variable de l'argument qu'on passe :

```
>>> def sub(a, b):
...     return a - b
...
>>> sub(3, 2)
1
>>> sub(a=3, b=2)
1
>>> sub(b=3, a=2)
-1
```

Portée des variables

En Python, les variables sont locales aux fonctions.

```
>>> s = 3
>>> def square(n):
...     s = n ** 2
...     return s
...
>>> s
3
>>> square(3)
9
>>> s
3
```

Avez-vous des questions ?

Travaux pratiques

Instructions

Bases du langage Python

Traitement des erreurs

Introduction

Python gère les erreurs avec la structure de contrôle `try`/`except` et une hiérarchie de classes d'erreur.

Hiérarchie d'erreurs natives

La documentation officielle contient une liste complète.

Capturer une erreur

```
>>> def div(a, b):
...     try:
...         return a / b
...     except ArithmeticError:
...         return float("nan")
...
>>> div(4, 3)
1.3333333333333333
>>> div(4, 0)
nan
```

Lancer une erreur

```
>>> wind_speed = -9999
>>> if wind_speed < 0:
...     raise ValueError("Speed cannot be negative")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: Speed cannot be negative
```

Définir une erreur personnalisée

Peuvent hériter d'`Exception` ou d'une erreur plus précise.

```
class MathException(Exception):
    pass
```

Re-lancer une erreur

```
>>> try:
...     1 / 0
... except ArithmeticError:
...     print("Division impossible")
...     raise
...
Division impossible
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

Envelopper des erreurs

```
>>> class MathException(Exception):
...     pass
...
>>> try:
...     1 / 0
... except ArithmeticError as e:
...     raise MathException("Division impossible") from e
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
__main__.MathException: Division impossible
```

finally

Le mot-clef **finally** permet d'exécuter des instructions quoi qu'il arrive.

La documentation officielle contient plus de détails sur cette structure.

```
>>> try:
...     1 / 0
... finally:
...     print("Hello World")
...
Hello World
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

`else`

Comme pour les boucles, la clause `else` est exécutée si on sort normalement du `try`/`except` (sans passer par le `except`).

Avez-vous des questions ?

Bases du langage Python

Fonctions natives

Introduction

Certaines fonctions ne nécessitent pas d'import. Elles sont toujours disponibles.

Leur liste complète est disponible dans la documentation Python.

Création et conversion de types natifs

```
>>> int(42.0)
42

>>> str()
''
```

Mécanismes d'itération

```
>>> list(enumerate(["a", "b", "c"]))
[(0, 'a'), (1, 'b'), (2, 'c')]

>>> list(zip(["a", "b", "c"], [10, 20, 30]))
[('a', 10), ('b', 20), ('c', 30)]

>>> list(map(float, [1, 2, 3]))
[1.0, 2.0, 3.0]

>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(reversed([1, 2, 3]))
[3, 2, 1]

>>> sorted([3, 1, 2])
[1, 2, 3]
```

Mécanismes d'agrégation

```
>>> all([0, 42])
False

>>> any([0, 42])
True

>>> sum([1, 2, 3])
6

>>> max([1, 2, 3])
3

>>> min([1, 2, 3])
1
```

Manipulation d'objets

```
>>> class A:
...     pass
...
>>> a = A()

>>> setattr(a, "x", 3)
>>> a.x
3

>>> getattr(a, "x")
3

>>> delattr(a, "x")
>>> a.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'x'
```

Test d'appartenance à une classe

```
>>> callable(int)
True
>>> callable(42)
False

>>> type(True)
bool

>>> isinstance(True, int)
True
>>> type(True) == int
False

>>> issubclass(bool, int)
True
```

Opérations de base

```
>>> abs(-42)
42

>>> divmod(10, 3)
(3, 1)

>>> pow(2, 3)
8

>>> pow(3, 2, 4)
1
```

Méthode d'apprentissage

Les fonctions natives, comme le reste de la bibliothèque standard, s'apprennent au fil des projets :

- En code review, soyez curieu·x·se
- Consultez la documentation régulièrement

Avez-vous des questions ?

Éléments de Python avancé

Éléments de Python avancé

Générateurs

Introduction

Buts :

- Transformer des données sans stocker le résultat en mémoire
- Donc traiter de grands volumes de données facilement
- Remplacer avantageusement les listes pour certaines utilisations
- Plus simples à créer qu'un itérateur classique (supertype de générateur)
- Offre des possibilités avancées de communication

Intuition

Les générateurs stockent un algorithme pour fabriquer des données.

Cet algorithme n'est utilisé que lorsque la valeur est demandée, pas avant (*lazy evaluation*).

Hiérarchie de l'itération en Python

Du plus général au plus spécifique :

Iterable Méthode `__iter__`

Iterator Méthodes `__iter__` et `__next__`

Generator Méthodes `send`, `throw`, `close`, `__iter__` et `__next__`

Ces méthodes doivent implémenter les contrats suivants :

`__iter__` Renvoie un itérateur

`__next__` Demande l'objet suivant à un itérateur

Nous verrons plus loin ce que font `send`, `throw` & `close`.

Création d'un générateur

Deux options :

- Par compréhension
- Avec une fonction et les mots clefs `yield` et `yield from`

Exemple — Par compréhension

```
total = 0
generator = (n ** 2 for n in range(1_000_000_000))
for i in generator:
    total += i
print(total)
```

Exemple — Par une fonction utilisant yield

```
def double(limit):
    for i in range(limit):
        yield i * 2


total = 0
for i in double(1_000_000_000):
    total += i
print(total)
```

Fonctions qui acceptent des générateurs

Beaucoup de fonctions acceptent des générateurs. Quand c'est le seul argument, pas besoin de parenthèses :

```
sum(n ** 2 for n in range(1000))
```

Utilisation d'un générateur

Comme pour un itérateur :

- Itération à l'aide d'une boucle `for`

```
for i in generator:
    print(i)
```

- Manuellement avec la méthode `next`

```
item = next(generator)
```

- Utilisation unique, une fois épuisé, il faut recréer l'itérateur pour ré-itérer

Chaînage de générateurs

```
def squares(iterator):
    for item in iterator:
        yield item ** 2


def odd(iterator):
    for item in iterator:
        if item % 2:
            yield item


for i in squares(odd(range(10))):
    print(i)
```

Ou avec une compréhension :

```
for i in (n ** 2 for n in (n for n in range(10) if n % 2)):
    print(i)
```

Fonctions avancées

`yield` retourne les valeurs qu'on lui envoie avec `send` :

```
def squares():
    while True:
        value = (yield) ** 2
        print(value)


generator = squares()
next(generator)
for i in range(10):
    generator.send(i)
```

`next(generator)` est équivalent à `generator.send(None)`

De la même manière, les méthodes `throw` et `close` envoient des exceptions au point de pause.

Mot clef `yield from`

Le mot clef `yield from` permet de `yield` depuis un autre générateur :

```
def integers(n):
    for i in range(n):
        yield i


def integers_repeated(n, repeat_n):
    for i in range(repeat_n):
        yield from integers(n)


for i in integers_repeated(3, 2):
    print(i)
```

Avez-vous des questions ?

Travaux pratiques

Instructions

Éléments de Python avancé

Programmation orientée objet

Éléments de Python avancé

Programmation orientée objet

Concepts fondamentaux

Qu’est-ce qu’un paradigme ?

Un paradigme de programmation consiste en une manière d’appréhender un problème informatique.

Plus pragmatiquement, cela décrit la manière de structurer du code et le fonctionnement d’un langage de programmation.

Quelques paradigmes classiques

Parmi les paradigmes les plus fréquents, on retrouve les paradigmes suivants:

- **Impératif** : Le plus classique. La structuration du code suit l'ordre d'exécution des instructions.
- **Fonctionnel** : Héritage du lambda-calcul. Un programme informatique est vu comme une fonction au sens mathématique. Une fonction peut prendre une autre fonction en paramètre.
- **Logique** : Un programme informatique est vu comme un ensemble de propositions logiques et de prédicats.

⇒ **Orienté objet** : Un programme est structuré en un ensemble de classe. *C'est celui sur lequel nous allons nous attarder dans ce cours.*

Paradigmes et langages

Historiquement, un langage de programmation suivait un unique paradigme de programmation. Aujourd'hui, de nombreux langages en implémentent plusieurs.

Néanmoins, chaque langage a souvent son paradigme principal.

Le paradigme objet

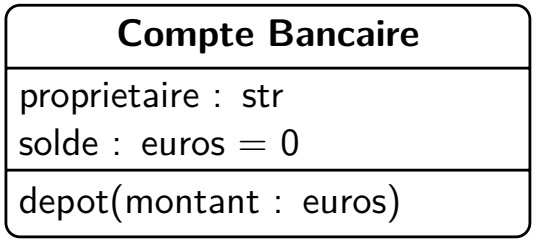
Avec l'orienté objet, un programme est un ensemble de classes qui décrivent différentes fonctionnalités, éléments ou données de celui-ci.

L'exécution d'un programme correspond à l'ensemble des interactions entre ces classes.

Qu'est-ce qu'une classe ?

La classe est l'élément structurant du programme. Une classe est décomposée en deux parties :

- les **attributs** qui sont l'ensemble des variables que la classe manipule.
- les **méthodes** qui sont l'ensemble de fonctions qui permettent de manipuler ses attributs.



Une classe représentant un compte bancaire, F.-M. Giraud, R. Rincé & H. Mougard, CC-BY-SA-4.0.

Cette organisation des données dans des classes est l'un des concepts fondateurs de l'orienté objet permettant l'encapsulation.

Classes et Objets

Une classe représente une structure de données dotée de méthodes pour la manipuler, mais pour l'utiliser il faut d'abord l'instancier.

Instancier une classe consiste à créer l'élément que représente la classe en mémoire et pouvoir l'utiliser. Cette instanciation se fait en utilisant un *constructeur* de la classe. L'instance de la classe qui sera créée s'appelle un objet.

Un objet est la représentation concrète, manipulable et stockée en mémoire d'une classe.

Classes et Objets

Points importants :

- Il est possible de créer plusieurs objets à partir d'une même classe.
- Chaque objet créé à partir d'une même classe dispose de ses propres attributs.
- Une classe définie la structure d'un objet. On peut donc dire que la classe d'un objet est aussi son type.

Modificateurs d'accès

En théorie, les attributs et les méthodes peuvent avoir différents modificateurs d'accès. On en trouve traditionnellement trois du plus restrictif au moins restrictif :

- privé (*private*) : Accessible uniquement par la classe qui a définie la méthode privée (pas juste son prototype).
- protégé (*protected*) : Accessible par la classe qui a définie la méthode protégée et par toutes les classes qui en héritent.
- public : Accessible par tout le monde

Modificateurs d'accès : getters et setters

Conceptuellement, pour garantir l'encapsulation, les attributs sont privés ou protégés, mais jamais publiques pour éviter la corruption d'état.

Il faut alors des méthodes spécifiques d'accès pour les attributs. En POO, ces méthodes sont appelées getters et setters.

Selon l'accès souhaité, ces méthodes seront publiques ou protégées.

Concepts structurants

Au-delà de l'encapsulation, la POO (*Programmation Orienté Objet*) est structurée autour de deux autres concepts fondamentaux :

- Héritage
- Polymorphisme

Héritage

L'héritage consiste à définir une nouvelle classe à partir d'une classe existante.

La nouvelle classe ainsi créée *hérite* des mêmes attributs et méthodes que la classe initiale.

On parle alors de classe *enfant* ou *fille* pour la nouvelle classe et de classe *parente* ou *mère* pour la classe initiale.

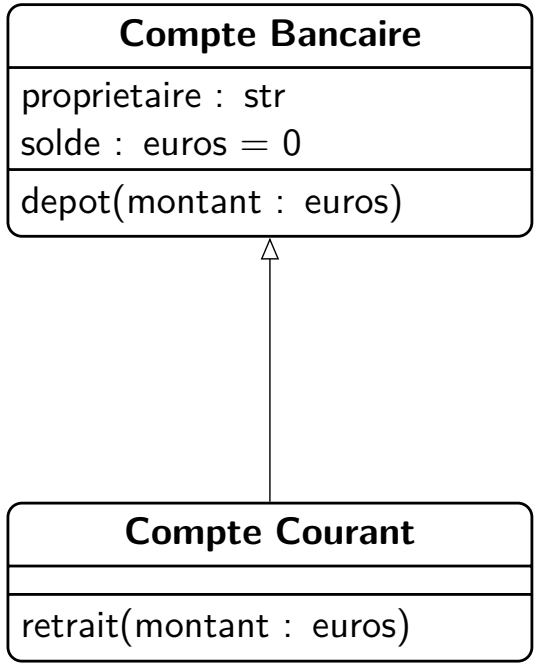


Diagramme UML représentant un héritage, F.-M. Giraud, R. Rincé & H. Mougard, CC-BY-SA-4.0.

Héritage

Il est possible de créer autant de classes filles que souhaité.

Chaque nouvelle classe créée dispose en général d'attributs et de méthodes qui lui sont propres et qui permettent de spécifier son comportement particulier.

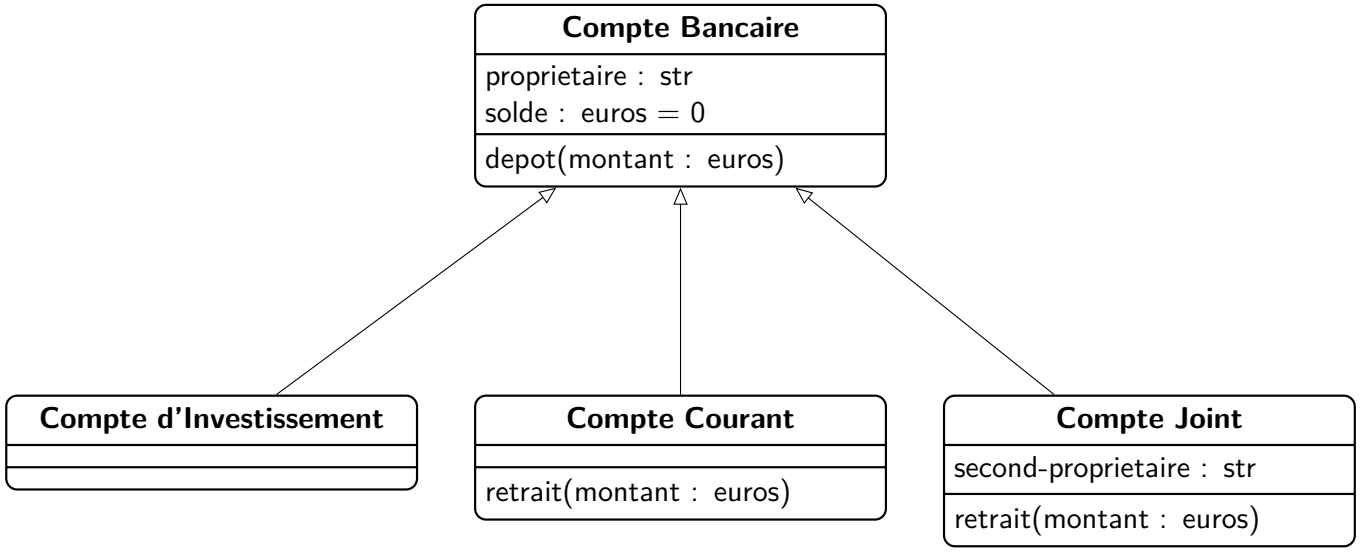


Diagramme UML représentant trois héritages, F.-M. Giraud, R. Rincé & H. Mougard, CC-BY-SA-4.0.

Héritage multiple

Il est possible qu'une classe hérite de plusieurs classes permettant ainsi de composer une classe regroupant les fonctionnalités de plusieurs classes.

Si cela est autorisé, l'héritage multiple apporte aussi de nombreuses contraintes de développement et de complexité qui peuvent rendre l'écriture du code particulièrement difficile.

Certains langages l'interdisent (tel que Java).

Héritage

L'héritage permet aussi d'utiliser le dernier concept fondamental de la POO qu'est le polymorphisme.

Polymorphisme

Le polymorphisme regroupe en fait plusieurs concepts qui ne dépendent pas forcément du paradigme objet. Pour simplifier, il est possible de le décomposer en deux sous-concepts :

- La surcharge de méthodes/fonctions
- Le sous typage

Polymorphisme : Surcharge

La surcharge permet de modifier le comportement d'une fonction selon le type des attributs qui lui sont passés.

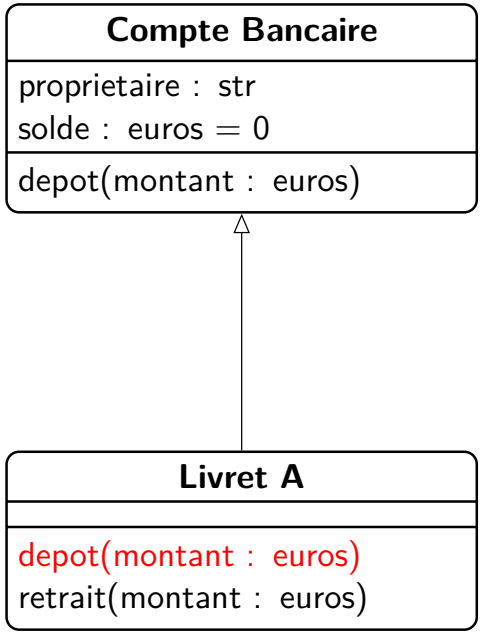
Sans parler d'objet, un exemple classique est la fonction d'addition traditionnellement notée **+**

```
>>> # (int, int) -> int
>>> 3 + 5
8
>>> # (float, float) -> float
>>> 1.0 + 3.0
4.0
>>> # (float, int) -> float
>>> 7.5 + 3
10.5
>>> # (str, str) -> str
>>> "abc" + "def"
'abcdef'
```

Polymorphisme : Surcharge

Dans le cas de la POO, la surcharge de fonction se fait généralement entre les méthodes d'une classe parente et d'une classe fille.

Dans l'exemple ci-contre, la méthode `depot` dans la classe `Compte Bancaire` consiste juste à ajouter une somme d'argent a la variable `solde`.
Dans la classe `Livret A`, il n'est pas possible d'ajouter de l'argent si le solde atteint un certain seuil. Il faut donc surcharger cette méthode.



Polymorphisme : Sous-typage

Le sous-typage permet d'utiliser une méthode ou fonction qui prend en paramètre une autre classe que celle initialement définie du moment qu'elle en soit une descendante.

Polymorphisme : Sous-typage

Supposons la fonction :
`somme_comptes(comptes : list[Compte Bancaire]) -> euros`
qui calcule la somme d'une liste de compte bancaire.

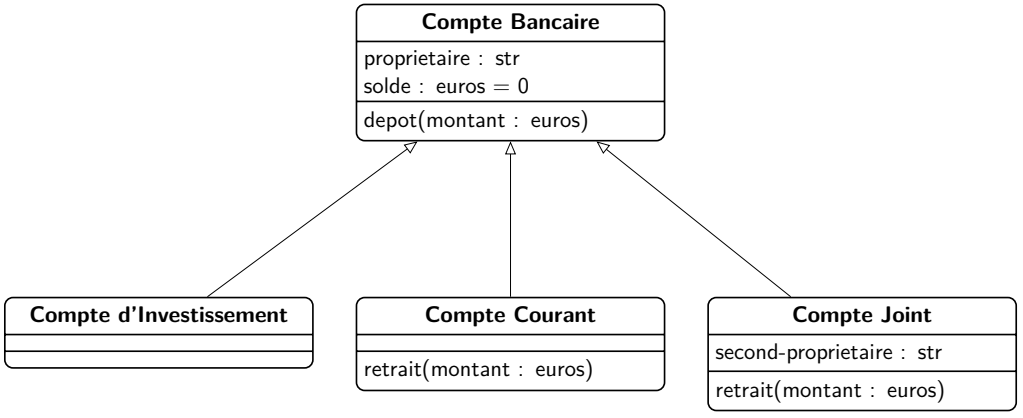


Diagramme UML représentant trois héritages, F.-M. Giraud, R. Rincé & H. Mougard, CC-BY-SA-4.0.

Étant donné que `Compte Courant`, `Compte Joint`, etc. héritent de `Compte Bancaire`, on peut utiliser la fonction `somme_comptes` pour additionner les soldes indépendamment de la classe réelle des comptes bancaires.

Polymorphisme

Il s'agit de quelques exemples de polymorphismes, mais il est tout à fait possible de trouver des cas plus complexes qui utilisent à la fois de la surcharge et du sous-typage.

Interface et classes abstraites

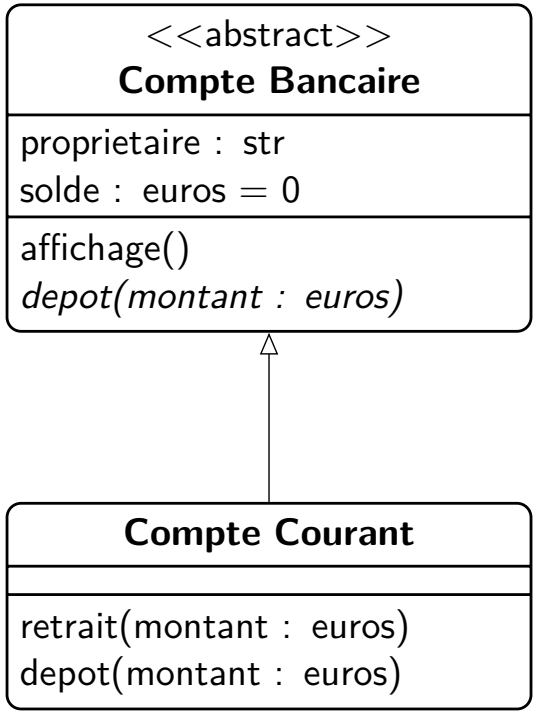
Dans la POO, il existe aussi deux types de structures assez classiques :

- Une interface est un squelette de classe qui ne contient pas de code. Elle ne contient que les prototypes des méthodes et parfois le nom des attributs (selon le langage).
- Une classe abstraite se situe à mi-chemin entre une interface et une classe. Celle-ci peut contenir des méthodes avec du code, mais certaines méthodes restent juste des prototypes.

Dans tous les cas, les méthodes qui n'ont pas été implémentées devront l'être dans toutes les classes qui héritent de la classe abstraite ou de l'interface.

Exemple de classe abstraite

L'exemple ci-contre montre une classe `Compte Courant` qui hérite de la classe abstraite `Compte Bancaire`. Celle-ci contient des méthodes définies et des prototypes de méthodes (*en italique*).

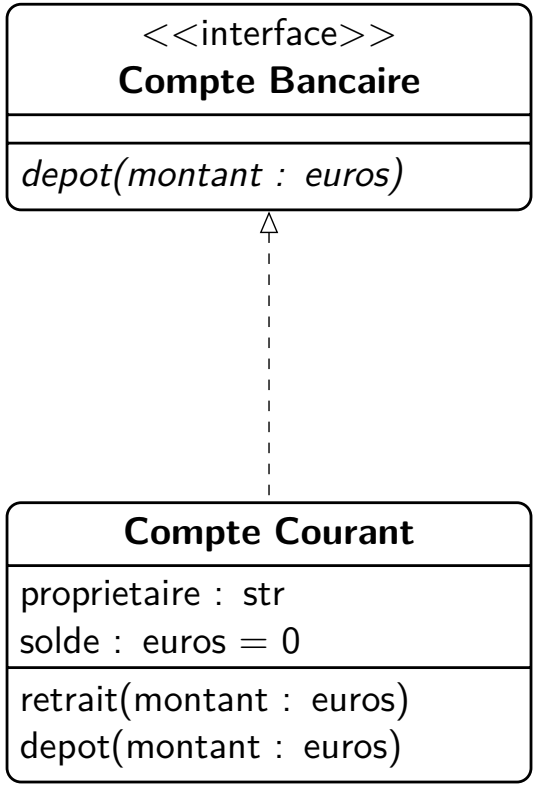


Exemple de classe abstraite, F.-M. Giraud, R. Rincé & H. Mougard, CC-BY-SA-4.0.

Exemple d'interface

Même cas avec une interface. On dit que la classe **implémente** l'interface.
La possibilité d'avoir des attributs dans les interfaces dépend du langage.

Pour l'exemple, les attributs ont été déplacés dans la classe fille. (Java ne permet pas d'avoir d'attribut dans une interface, mais C++ l'autorise).



Exemple d'une interface, F.-M. Giraud, R. Rincé & H. Mougard, CC-BY-SA-4.0.

Lexique

- **Classe** : Une structure de données dotée de méthodes et d'attributs.
- **Méthode** : Une fonction associée à une classe.
- **Attribut** : Une variable associée à une classe.
- **Objet** : Une instance d'une classe qui existe en mémoire à un moment donnée.
- **Encapsulation** : Principe de gestion des données d'un programme qui consiste à masquer les données brutes via des fonctions d'accès.
- **Héritage** : Mécanisme permettant de définir une classe à partir d'une autre.
- **Polymorphisme** : Mécanismes permettant d'utiliser une méthode sur différents types de variables ou de faire passer un objet pour un autre.

Lexique

- **Constructeur** : Méthode dédiée permettant de créer un objet à partir d'une classe.
- **Getter** : Méthode d'accès à un attribut.
- **Setter** : Méthode de modification d'un attribut.
- **Interface** : Un squelette de classe qui ne contient pas de code, seulement les prototypes de méthodes.
- **Classe abstraite** : Une classe partiellement définie.

Éléments de Python avancé

Programmation orientée objet

Programmation orientée objet en Python

POO en Python

Python est un langage multi-paradigmes qui utilisent principalement le paradigme impératif et le paradigme objet.

L'implémentation de l'orienté objet en python est moins restrictif que des langages tels que Java ou C++, s'autorisant un certain nombre de libertés.

Définition d'une classe : Structure de base

- Une classe se définie avec le mot clef `class` suivit de son nom. Par convention, le nom de la classe commence par une majuscule.
- Le constructeur de la classe est défini avec la méthode `__init__` qui est appelée automatiquement lors de l'instanciation d'un objet.

```
from math import sqrt


class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        return sqrt((other.x - self.x) ** 2 + (other.y - self.y) ** 2)
```

Définition d'une classe : `self`

- Le mot clef **`self`** est un attribut d'instance qui pointe vers l'objet qui instancie la classe et utilise une de ses méthodes. Il permet d'accéder aux attributs et aux méthodes de la classe.
- Les attributs d'une classe sont définis dans le constructeur avec le mot clef **`self`** et une affectation de valeur.
- Le mot clef **`self`** est toujours le premier paramètre d'une méthode de classe.

```
from math import sqrt


class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, other):
        return sqrt((other.x - self.x) ** 2 + (other.y - self.y) ** 2)
```

Instanciation

- L'instanciation d'une classe se fait en utilisant le nom de la classe suivit des paramètres définis dans le constructeur (`__init__`).
- Le paramètre `self` est automatiquement fourni en tant que premier paramètre. Il ne faut donc pas le fournir lors de l'appel.

```
>>> from math import sqrt
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...     def distance(self, other):
...         return sqrt((other.x - self.x) ** 2
...                     + (other.y - self.y) ** 2)
...
>>> point1 = Point(2, 2)
>>> point2 = Point(-2, -2)
>>> point1.distance(point2)
5.656854249492381
```

Modificateurs d'accès

Sans spécification particulière, les attributs et méthodes d'une classe sont publics.

Python n'implémente que partiellement les modificateurs d'accès *protégé* ou *privé*.

Modificateurs d'accès : Syntaxe

La notion de modificateurs est principalement définie par la PEP 8.
Elle y définie deux syntaxes :

- `_nom_variable` avec un « _ » devant le nom de la variable ou méthode.
- `__nom_variable` avec **deux** « _ » devant le nom de la variable ou méthode.

Modificateurs d'accès : `_nom_variable`

La notation _nom_variable indique que la variable (ou méthode) n'est pas publique et qu'elle ne doit pas être accédée hors de la classe.

C'est ce qui se rapproche le plus de *protégé*, mais python n'empêche pas de l'accès. Il s'agit juste d'une convention.

Modificateurs d'accès : `__nom_variable`

La notation `__nom_variable`, contrairement, à la précédente, n'est pas juste une convention et produit un effet en modifiant le nom de la variable pour tous les accès qui ne sont pas la classe en elle-même.

Ce mécanisme s'appelle le *name mangling* (mutilation de nom). C'est ce qui s'apparente le plus à un modificateur *privé*, mais il n'est pas vraiment utilisé ainsi.

Cette notation reste très controversée

Exemple de modificateurs d'accès

```
>>> class Point:
...     def __init__(self, x, y):
...         self._x = x # Un seul underscore
...         self.__y = y # Deux underscores
...
>>> point = Point(3, 4)
>>> point._x # Possible mais considéré comme une faute
3
>>> point.__y
AttributeError: 'Point' object has no attribute '__y'
>>> point._Point__y # Name mangling
4
```

Héritage

L'héritage en python est syntaxiquement simple.

Lors de la spécification d'une nouvelle classe, la classe parente est spécifiée entre parenthèses après le nom de la nouvelle classe.

```
>>> class Rectangle:
...     def __init__(self, width, height):
...         self.width = width
...         self.height = height
...     def area(self):
...         return self.width * self.height
...
>>> class Square(Rectangle):
...     def __init__(self, side):
...         super().__init__(side, side)
...
>>> Square(2).area()
4
```

Héritage

Le mot clef **`super()`** est une méthode qui pointe sur la classe parente et qui permet notamment d'accéder à son constructeur.

Généralement, le constructeur de la classe parente est appelé en première ligne du constructeur de la nouvelle classe.

```
>>> class Rectangle:
...     def __init__(self, width, height):
...         self.width = width
...         self.height = height
...     def area(self):
...         return self.width * self.height
...
>>> class Square(Rectangle):
...     def __init__(self, side):
...         super().__init__(side, side)
...
>>> Square(2).area()
4
```

Héritage

Il est possible d'éviter le mot clef `super` en utilisant une syntaxe particulière.

Il faut alors spécifier la classe parente et fournir le paramètre self au constructeur de la classe parente.

```
>>> class Rectangle:
...     def __init__(self, width, height):
...         self.width = width
...         self.height = height
...     def area(self):
...         return self.width * self.height
...
>>> class Square(Rectangle):
...     def __init__(self, side):
...         Rectangle.__init__(self,side, side)
...
>>> Square(2).area()
4
```

Cette syntaxe peut être nécessaire dans le cas de l'héritage multiple.

Méthodes __<nom méthode>__

Les classes python disposent, par défaut, d'un certain nombre de méthodes de la forme __<nom méthode>__, parfois déjà implémentées. Cette forme de nommage est réservée à ces méthodes particulières.

Par exemple, `__init__` en est l'une d'elle et définit le comportement du constructeur de classe.

Une liste exhaustive est détaillée dans la documentation officielle. Nous allons néanmoins en présenter quelques-unes des plus usuelles.

Surcharge d'opérateurs

Tous les opérateurs classiques du langage (`+`, `-`, `*`, `/`, `**`, `==`, `!=`, `<`, `<=`, `>`, `>=`, etc.) peuvent être implémentés avec un comportement spécifique pour une classe donnée.

Chaque opérateur a sa méthode dédiée : `__add__` pour l'opérateur `+`, `__eq__` pour `==`, etc.

```
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...     def __add__(self, other):
...         return Point(self.x + other.x, self.y + other.y)
...
>>> point1 = Point(2, 3)
>>> point2 = Point(3, 4)
>>> point3 = point1 + point2
>>> point3.x, point3.y
(5, 7)
```

Autres méthodes spécifiques

Il existe un large éventail de méthodes spécifiques plus ou moins complexes telles que:

- `__str__()` pour manipuler le comportement de la fonction de cast en chaine de caractères `str()` sur un objet.
- `__bool__()` pour manipuler le comportement de la fonction de cast en booléen `bool()` sur un objet.
- `__len__()` pour manipuler le comportement de la fonction `len()` qui renvoie usuellement le nombre d'éléments dans un itérable.
- `__iter__()` qui renvoie un itérateur (notamment utilisé par la boucle `for`)
- ... ⇒ Documentation officielle

Interfaces et classes abstraites en python

Il n'y a pas d'interface en python.

Il faudra se contenter de classes abstraites.

Classes abstraites en python

L'implémentation d'une classe abstraite en python nécessite l'usage de la librairie `abc` (*Abstract Base Class*) de la bibliothèque standard.

La classe abstraite à définir doit hériter de la classe `abc.ABC`

```
>>> from abc import ABC, abstractmethod
>>> class Polygon(ABC):
...   @abstractmethod
...   def area(self)->float|int:
...     """Compute the area of the polygon"""
...     pass
...
>>> class Rectangle(Polygon):
...   def __init__(self, width, height):
...     self.width = width
...     self.height = height
...   def area(self):
...     return self.width * self.height
>>> r = Rectangle(4,3)
>>> r.area()
12
```

Classes abstraites en python

Les prototypes de méthodes sont indiqués avec le décorateur `@abstractmethod`. Celui-ci indique que la méthode suivante doit être implémentée dans les classes qui hériteront de la classe abstraite.

```
>>> from abc import ABC, abstractmethod
>>> class Polygon(ABC):
...   @abstractmethod
...   def area(self)->float|int:
...     """Compute the area of the polygon"""
...     pass
...
>>> class Rectangle(Polygon):
...   def __init__(self, width, height):
...     self.width = width
...     self.height = height
...   def area(self):
...     return self.width * self.height
>>> r = Rectangle(4,3)
>>> r.area()
12
```

Classes abstraites en python

Si une classe hérite de la classe abstraite, mais oublie d'implémenter une des méthodes abstraites, cela renvoie une erreur.

```
>>> from abc import ABC, abstractmethod
>>> class Polygon(ABC):
...   @abstractmethod
...   def area(self)->float|int:
...     """Compute the area of the polygon"""
...     pass
...
>>> class Rectangle(Polygon):
...   def __init__(self, width, height):
...     self.width = width
...     self.height = height
>>> r = Rectangle(4,3)
TypeError: Can't instantiate abstract class Rectangle
with abstract method area
```

Avez-vous des questions ?

Travaux pratiques

Instructions

Éléments de Python avancé

Vérifications en Python

Introduction

Buts :

- Respect des normes dans une équipe
- Minimiser les pertes de temps en revue de code
- Produire un logiciel plus maintenable

Possibilités

Les points majeurs que l'on vérifie habituellement :

- Types
- Éléments de code non utilisés
- Documentation non présente ou incomplète
- Motifs qui sont souvent des bugs
- Style du code

Outils

L'organisation PyCQA maintient plusieurs outils de vérification, parmi eux :

- `flake8`
- `pylint`
- `RedBaron`

On utilise aussi beaucoup `mypy` pour vérifier les types & `black` pour le formatage.

Typage

Sûrement le point le plus important à vérifier :

- Permet de modifier aggressivement le logiciel
- Résout une grande partie des bugs avant l'intégration à la base de code
- Documente le code
- Permet des générations automatiques (CLI avec Typer par exemple)

Le passage à une base de code typée peut être progressif.

Notations de typage

On type une expression en précédant l'annotation de type de `:`.

Pour les retours de fonctions, on utilise `->`.

```
class TargetBuilder:
    def _parse_section_dir(
        self, section_path: str, custom_config: Dict[str, Any],
    ) -> Optional[Tuple[Section, Dependencies]]:
        ...
```

Des types additionnels sont présents dans le module `typing`.

Vérifications générales

L'outil incontournable est `flake8`. En fonction des extensions installées, il vérifiera :

- La syntaxe
- Les éléments non utilisés
- Les imports
- La complexité du code
- …

Extensions de `flake8`

Quelques extensions très pratiques :

- `flake8-bugbear`
- `flake8-docstrings`
- `flake8-import-order`
- `flake8-black`
- `flakehell`

Retrouvez une liste plus complète dans ce dépôt `git`.

Configuration de `flake8`

`flake8` se configure dans le fichier `.flake8` à la racine de votre projet.

```
[flake8]
ignore=B008,E121,E123,E126,E203,E226,E24,E704,W503,W504,D100,D105,D200
max-line-length=88
exclude=
    .git
import-order-style=appnexus
application-package-names=deckz
per-file-ignores=
    **/tests/*:D
    **/test*.py:D
max-complexity=9
```

Vérification du style

Pour le style, je vous recommande fortement l'utilisation de `black` :

- Formatage automatique (transformation du code source en arbre syntaxique puis de nouveau en code source ⇒ formatage original inutilisé)
- Possibilité de vérifier l'application sur la base de code
- Style homogène
- Plus de problèmes de style en revue de code !

Vérification de la documentation

Utilisation de `pylint` avec seulement ces options :

- `missing-param-doc`
- `differing-param-doc`
- `differing-type-doc`
- `missing-return-doc`

Ainsi que `flake8-docstrings` pour le style des docstrings.

Préconisations d'utilisation

Recommandé : regrouper les vérifications dans une seule commande (avec un Makefile par exemple).

- Faire tourner ces vérifications avant chaque PR (hook git possible)
- Les intégrer à la CI
- Une PR ne doit pas être revue tant que ces soucis ne sont pas réglés

Paramétrage

- Définir en équipe les points à vérifier
- Analyser régulièrement le coût/bénéfice de chaque élément vérifié
- Rajouter ou enlever des vérifications en fonction de ces analyses

Reproductibilité

Il faut gérer les outils de vérification comme des dépendances :

- Chaque collègue doit avoir les mêmes résultats étant donné une base de code
- Les outils doivent être faciles à installer, y compris sur la CI

Attention toutefois à ne pas installer ces dépendances en prod.

Avez-vous des questions ?

Travaux pratiques

Instructions

Éléments de Python avancé

Ingénierie de projets en Python

Éléments de Python avancé

Ingénierie de projets en Python

Gestion de l'environment logiciel

Introduction

Contrôler son environnement logiciel pour :

- Rendre son environnement de développement reproductible
- Contrôler les dépendances pour la prod
- Déployer son environnement facilement sur différents clouds

Isolation de l'environnement — venv

venv permet d'isoler un environnement Python :

- N'intéragit pas avec l'environnement système
- Permet la cohabitation de plusieurs environnements incompatibles
- Plus rapide et natif que les solutions basées sur les conteneurs
- Copie d'une distribution Python de base + customisation

Utilisation de venv

```
$ python3 -m venv .venv
$ source .venv/bin/activate
(.venv) $ pip install requests
```

Contrôle des dépendances

But : décrire précisément quelles libraires (et leur version) sont utilisées.

Plusieurs bons outils :

`pip` + `setup.py` Outil traditionnel, définit un paquet Python

`pip` + `requirements.txt` Liste simple de dépendances

`pip` + `pip-compile` Lister les dépendances puis les geler pour la stabilité

`Pipenv` Définition moderne d'une **application** Python (pas librairie)

`poetry` Définition moderne d'application ou librairie

Utilisation de `poetry`

Quelques commandes utiles pour la gestion de dépendances :

`poetry new` Crée un squelette de paquet Python géré par **`poetry`**

`poetry add` Ajoute un paquet comme dépendance du projet

`poetry remove` Enlève un paquet comme dépendance du projet

`poetry install` Installe les dépendances telles que spécifiées dans le fichier **`poetry.lock`** (géré automatiquement par **`poetry`**)

`poetry export` Exporte le fichier **`pyproject.toml`** vers d'autres formats (comme **`requirements.txt`**)

Faciliter le déploiement

Quelques plate-formes minoritaires acceptent les paquets Python pour le déploiement.

Le plus souvent, une étape intermédiaire est nécessaire avant le déploiement, la transformation en conteneur :

- Permet de contrôler les librairies natives en plus des librairies Python
- Plus grande robustesse si le logiciel s'exécute sur plusieurs OS
- Plus lourd à mettre en place que `venv`
- Plus adapté pour la prod (déploiement facile par Kubernetes)

⇒ Dockerisation du paquet Python à l'aide d'un fichier `Dockerfile`

Exemple de `Dockerfile`

```
FROM python:3.8.7-slim-buster as builder
WORKDIR /app
RUN apt update \
  && apt install -y --no-install-recommends \
  curl build-essential python3-dev gfortran libgl1-mesa-glx \
  libopenblas-dev liblapack-dev \
  && curl -sSL \
  https://raw.githubusercontent.com/python-poetry/poetry/master/\
  get-poetry.py | python
COPY poetry.lock ./
COPY pyproject.toml ./
RUN /root/.poetry/bin/poetry install --no-dev
COPY . .
RUN /root/.poetry/bin/poetry install --no-dev
CMD ["/root/.poetry/bin/poetry", "run", "landscapred"]
```

Avez-vous des questions ?

Démonstration

Les différents systèmes vus pendant la démonstration sont :

- `pip install -r requirements.txt` et `pip freeze` pour geler les versions
- `venv` et `virtualenv`, deux systèmes d'environnements virtuels
- `virtualenvwrapper`, un ensemble de scripts qui facilite la gestion des environnements virtuels
- `poetry`, l'outil que je recommande pour gérer les environnements en Python

Éléments de Python avancé

Ingénierie de projets en Python

Création de paquets Python

Introduction

Buts :

- Pouvoir publier un logiciel Python sous forme de paquet sur PyPI
- Bien définir ses dépendances et méta-données

Paquet Python

Pour transformer un dossier contenant des sources Python en paquet :

- Rajouter des fichier `__init__.py` dans chaque dossier de source
- Ajouter des méta-données (dépendantes de l'outil de build)

Outils

- `setup.py`, & `setuptools`
- `Pipenv`
- `poetry`

`poetry` est recommandé actuellement. Outil simple, moderne et complet.

Structure d'un paquet géré par `poetry`

```
demo-package
├── pyproject.toml
├── README.md
├── demo_package
│   ├── __init__.py
│   ├── module1.py
│   └── module2.py
└── tests
    ├── __init__.py
    ├── module1.py
    └── module2.py
```

Attention

Le nom du dossier source n'est pas `src` mais le nom du paquet (avec les - remplacés par des _)

Fichier `pyproject.toml`

Standard depuis la PEP 518 :

- Définit l'outil de build du paquet à utiliser
- Contient les méta-données
- Spécifie les dépendances, y compris de développement et extras
- En cours d'adoption par toute la chaîne d'outillage
- Déjà utilisable car supporté par `pip`

Exemple de fichier `pyproject.toml`

```toml
[tool.poetry]
authors = ["m09 <142691+m09@users.noreply.github.com>"]
description = "Tool to handle multiple beamer decks."
name = "deckz"
readme = "README.md"
version = "10.3.0"

[tool.poetry.dependencies]
python = "^3.9"
typer = {version = "^0.7.0", extras = ["all"]}

[tool.poetry.group.dev.dependencies]
pytest = "^7.2.1"

[tool.poetry.scripts]
deckz = 'deckz.cli:main'

[build-system]
build-backend = "poetry.core.masonry.api"
requires = ["poetry-core"]
```

Utilisation de `poetry`

Quelques commandes utiles pour la gestion de paquets :

`poetry version` Affiche ou modifie la version du paquet

`poetry build` Construit la distribution du paquet

`poetry publish` Publie le paquet sur PyPI

Dépôt PyPI

Dépôt des projets Python :

- Utilisation depuis `pip`, `poetry`, `Pipenv`, …
- Impossible de remplacer les fichiers d'une version par de nouveaux fichiers
- Possibilité d'utilisation depuis la CI

Avez-vous des questions ?

Travaux pratiques

Instructions

Éléments de Python avancé

Ingénierie de projets en Python

Interfaces en ligne de commande

Introduction

Buts :

- Créer des outils facilement manipulables
- Pouvoir scripter à partir de vos programmes Python

Mécanismes

- `sys.argv` contient les arguments passés à votre programme
- `argparse`, `click` ou encore `Typer` parsent ce tableau pour vous

`argparse`

Pour définir une ILC avec `argparse`, il faut :

- Créer un parser
- Appeler ce parser depuis le point d'entrée du programme
- Traiter les arguments parsés

Création d'un parser

Création d'un `argparse.ArgumentParser` puis ajout d'arguments avec la méthode `add_argument` :

```
import argparse

parser = argparse.ArgumentParser(description="Create datasets.")
parser.add_argument("output_dir", help="Path to the directory.")
parser.add_argument("--langs", nargs="+", help="Langs to consider.")
parser.add_argument("--max-size", type=int, help="Maximum size in ko.")
```

Ajouter un argument

On peut spécifier à `add_argument` de nombreux points. Parmi eux :

- Une option courte (`"-d"`) et longue (`"--directory"`)
- Une chaîne d'aide (`help="Aide de l'argument."`)
- Un type (`type=int`)
- Une action, pour créer des flags booléens par exemple
- Une valeur par défaut

La documentation officielle est très complète.

Utilisation

Appel de la méthode `parse_args` puis traitement des arguments recupérés.

```
args = parser.parse_args()
if args.command == "download":
    downloader = Downloader(args.output_dir)
    downloader.download_all_wiktionaries()
elif args.command == "dataset":
    create_csv_dataset_from_dump(args.dumps_folder_path)
```

Commandes multiples

Possibilité de définir plusieurs commandes : création des sous-parsers que l'on traite ensuite comme le parser principal :

```
import argparse

parser = argparse.ArgumentParser(description="Create datasets.")
subparsers = parser.add_subparsers(help="Commands", dest="command")

download_parser = subparsers.add_parser("download")
download_parser.add_argument("output_dir")

dataset_parser = subparsers.add_parser("dataset")
dataset_parser.add_argument("dumps_folder_path")
```

Utilisation avancée qui couple la CLI à la fonction appelée

Attention

Utilisation avancée, à regarder pendant un deuxième passage sur les slides.

```
args = parser.parse_args()
command = args.command
delattr(args, "command")
if command == "download":
    downloader = Downloader(**vars(args))
    downloader.download_all_wiktionaries()
elif command == "dataset":
    create_csv_dataset_from_dump(**vars(args))
```

- `vars` transforme l'objet `args` en dictionnaire
- `**` « éclate » le dictionnaire en mots-clefs et valeurs
- suppression préalable des arguments non-utilisés avec `delattr`

Quelques bonus apportés par `argparse`

- Aide générée automatiquement
- Messages d'erreurs informatifs quand les arguments sont incorrects

Avez-vous des questions ?

Travaux pratiques

Instructions

Fonctionnalités d’administration système

Fonctionnalités d'administration système

Manipulation de fichiers

Introduction

En Python, la manipulation de fichiers se fait principalement depuis 4 modules :

`shutil`	Opérations de haut niveau de copie et déplacement
`os`	Opérations de bas niveau
`pathlib`	Remplacement moderne pour une partie de **`os`**
`tempfile`	Création de fichiers et dossiers temporaires

Navigation · Module pathlib

```
>>> import pathlib

>>> formations_dir = pathlib.Path.home() / "work" / "formations"
>>> formations_dir
PosixPath('/home/mog/work/formations')
>>> str(formations_dir)
'/home/mog/work/formations'

>>> config_file = formations_dir / "config.yml"
>>> config_file.with_suffix(".txt")
PosixPath('/home/mog/work/formations/config.txt')
>>> config_file.name
'config.yml'

>>> for item in formations_dir.iterdir():
...     print(item)
...
/home/mog/work/formations/formation.sty
/home/mog/work/formations/.git
```

Utilisation de jokers

Il est possible d'utiliser des jokers grâce aux fonctions `pathlib.Path.glob` & `pathlib.Path.rglob` :

```
>>> import pathlib


>>> home = pathlib.Path.home()
>>> for jpg in home.rglob("*.jpg"):
...     print(jpg)
...
/home/mog/pictures/cat.jpg
/home/mog/pictures/dog.jpg
/home/mog/work/formations/img/python-logo.jpg
```

Reste de l'API `pathlib`

Consultez l'excellente documentation officielle pour découvrir l'API complète.

Copie, déplacement & suppression de fichiers et dossiers

`shutil.copy` Copie sans les méta-données

`shutil.copy2` Copie avec les méta-données

`shutil.copytree` Copie un dossier

`shutil.move` Déplace un fichier ou un dossier

`os.remove` Supprime un fichier

`shutil.rmtree` Supprime un dossier

Exemple d'utilisation de `shutil`

```
import pathlib
import shutil


home = pathlib.Path.home()
formations_dir = home / "formations"
backup_dir = home / "backup-formations"
old_backup_dir = home / "backup-formations-old"

if backup_dir.exists():
    shutil.move(backup_dir, old_backup_dir)

shutil.copytree(formations_dir, backup_dir)

if old_backup_dir.exists():
    shutil.rmtree(old_backup_dir)
```

Fichiers et dossiers temporaires

`tempfile.TemporaryFile` Crée un fichier temporaire

`tempfile.NamedTemporaryFile` Crée un fichier temporaire dont le nom est accessible (et pas seulement l'objet)

`tempfile.TemporaryDirectory` Crée un dossier temporaire

Toutes ces méthodes s'utilisent dans un bloc **`with`** pour s'assurer de l'effacement des fichiers/dossiers.

```
import tempfile


with tempfile.TemporaryFile(mode="w+", encoding="utf8") as fh:
    fh.write(template.render())
```

Création d'archives

`shutil.make_archive` crée des archives avec compression si les modules correspondant sont disponibles.

Manipulation de permissions, groupes et propriétaires

Le module `os` contient toutes ces fonctions quand elles sont exposées par le système d'exploitation.

Avez-vous des questions ?

Fonctionnalités d'administration système

Gestion de signaux

Introduction

Les signaux sont un mécanisme de communication entre processus hérité du C.

En particulier, les systèmes d'exploitation communiquent avec les programmes par ce mécanisme.

Module `signal` de la bibliothèque standard

En Python, le module `signal` est utilisé pour intercepter et envoyer des signaux. Il définit :

- Une variable pour chaque signal (`SIGINT`, `SIGTERM`, ...)
- Une fonction `signal` pour exécuter une fonction à chaque réception de signal
- Plusieurs fonctions connexes moins utilisées à étudier dans la documentation officielle

Signification de chaque signal

Description des signaux POSIX sur Wikipédia et des signaux du module `signal` dans la documentation Python.

Définition d'un gestionnaire de signal

```python
import os
import signal


def reload(signum, frame):
    print("Reloading config…")


signal.signal(signal.SIGUSR1, reload)
print(os.getpid())
while True:
    signal.pause()
```

```
$ python example.py &
16701
$ kill -USR1 16701
Reloading config…
$ kill -TERM 16701
[1]  + 16701 terminated  python example.py
```

Points à noter

- Les signaux diffèrent entre les systèmes d'exploitation
- Les signaux arrivent dans le thread principal
- On ne peut pas ignorer ou redéfinir `SIGKILL`
- Possibilités de situations de compétition entre signaux

Cadre d'utilisation

Conseillé

Traiter les signaux du système d'exploitation ou envoyés depuis des scripts d'admin ou des programmes C.
Suivre la norme POSIX pour réagir aux signaux.

Déconseillé

Pour faire de la communication à l'intérieur d'une application ou entre applications. Préférez :

- Exposer une interface RPC (*Remote Procedure Call*)
- Exposer un serveur HTTP
- Utiliser une queue d'événements
- ...

Avez-vous des questions ?

Fonctionnalités d'administration système

Variables d'environnement

Introduction

Les variables d'environnement sont clefs pour :

- Récupérer la configuration système
- Déployer facilement des applications Python
- Offrir une interface alternative à une CLI

Récupérer ou modifier une variable d'environnement

Deux options :

`os.environ` Dictionnaire avec valeurs textuelles

`os.environb` Dictionnaire avec valeurs binaires

Les variables d'environnement sont récupérées au moment de l'import du module `os`.

Avez-vous des questions ?

Fonctionnalités d’administration système

Flux standards

Introduction

Les flux d'entrée, de sortie et d'erreur sont des canaux de communication privilégiés pour les outils systèmes.

Accès en Python

Par le module `sys` :

- `sys.stdin`
- `sys.stdout`
- `sys.stderr`

Ces flux sont exposés comme des fichiers.

Gestion facile de `stdin` pour créer des outils système

Le module `fileinput` permet de créer des outils qui travaillent au choix sur :

- Les lignes de l'entrée standard
- Les lignes contenues dans les fichiers dont les noms sont données en argument

⇒ Définition d'outils standards type UNIX facile.

Redirection de des sortie et erreur standards

Le module `contextlib` propose deux fonctions utiles :

- `redirect_stdout`
- `redirect_stderr`

Avez-vous des questions ?

Fonctionnalités d’administration système

Commandes externes

Introduction

Le module `subprocess` permet :

- d'exécuter des commandes systèmes
- de mettre en place des *pipelines* comme on le ferait dans un terminal
- de récupérer les sorties des commandes au lieu de simplement les afficher
- de vérifier le code de retour

Et plus encore : documentation officielle

Exemple de code

```
>>> import subprocess
>>> subprocess.run(["grep",
...                 "linux-libc-dev",
...                 "/var/log/apt/history.log"])
...
Upgrade: linux-libc-dev:amd64 (5.15.0-105.115, 5.15.0-106.116)
CompletedProcess(args=['grep',
                       'linux-libc-dev',
                       '/var/log/apt/history.log'],
                 returncode=0)
```

Avez-vous des questions ?

Travaux pratiques

Instructions

Fonctionnalités d'administration système

Expressions régulières

Introduction

Capture et manipulation de chaînes de caractères par le biais de motifs génériques.

Exemple : rajouter un espace après chaque ponctuation

« Le chat mange,délicatement, ses croquettes.Jean est attendri. »

→

« Le chat mange,délicatement, ses croquettes. Jean est attendri. »

Utilisations

Une fois le motif capturé, on peut :

- supprimer
- modifier
- réutiliser
- ...

Caractères spéciaux

- `^` début de ligne (ou négation dans un ensemble)
- `$` fin de ligne
- `.` joker (capture n'importe quel caractère)
- `[` ouverture d'un ensemble de caractère
- `]` fermeture d'un ensemble de caractère
- `(` ouverture de groupe de capture
- `)` fermeture de groupe de capture
- `?` 0 ou 1 occurence du motif
- `*` de 0 à ∞ occurences du motif
- `+` de 1 à ∞ occurences du motif
- `{` ouverture d'intervalle de répétition du motif
- `}` fermeture d'intervalle de répétition du motif
- `\` caractère spécialement spécial
- `|` OU logique dans un groupe de capture

re.match

```python
import re


patterns = ["a.*s", "^b", "s$", "a+b", "a{4,5}b", "y*b"]
test_string = "baaabyys"
for pattern in patterns:
    if re.match(pattern, test_string):
        print(f"{pattern:10} a matché")
    else:
        print(f"{pattern:10} n'a pas matché")
```

```
a.*s       n'a pas matché
^b         a matché
s$         n'a pas matché
a+b        n'a pas matché
a{4,5}b    n'a pas matché
y*b        a matché
```

re.search

```python
import re


patterns = ["a.*s", "^b", "s$", "a+b", "a{4,5}b", "y*b"]
test_string = "baaabyys"
for pattern in patterns:
    if re.search(pattern, test_string):
        print(f"{pattern:10} a matché")
    else:
        print(f"{pattern:10} n'a pas matché")
```

```
a.*s       a matché
^b         a matché
s$         a matché
a+b        a matché
a{4,5}b    n'a pas matché
y*b        a matché
```

re.search

```python
import re


pattern = "a.y"
test_string = "baaabyys"
result = re.search(pattern, test_string)
print(f"Capture du caractère {result.start()} à {result.end()}")
print(f"Groupe capturé : {result.group()!r}")
```

```
Capture du caractère 3 à 6
Groupe capturé : 'aby'
```

re.findall

```python
import re


pattern = "b.."
test_string = "baaabyys"
result = re.findall(pattern, test_string)
print(result)
```

```
['baa', 'byy']
```

re.finditer

```python
import re


pattern = "b.."
test_string = "baaabyys"
result = re.finditer(pattern, test_string)
for r in result:
    print(f"{r.group()!r} de l'indice {r.start()} à {r.end()}")
```

```
'baa' de l'indice 0 à 3
'byy' de l'indice 4 à 7
```

Le caractère \

\ est un caractère de contôle spécial.
Permet de définir des groupes de caractères :

- **\d** numériques $\iff$ `[0-9]`
- **\D** non-numériques $\iff$ `[^0-9]`
- **\s** blancs $\iff$ `[ \t\n\r\f\v]`
- **\S** non-blancs $\iff$ `[^ \t\n\r\f\v]`
- **\w** alpha-numériques $\iff$ `[a-zA-Z0-9_]`
- **\W** non-alpha-numériques $\iff$ `[^a-zA-Z0-9_]`

Plus généralement \ permet de capturer les caractères spéciaux (ex: `\.` pour capturer le point)

Chaîne de caractère brute

Python utilise par défaut des chaînes de caractères littérale : \ est déjà un caractère spécial...

```
string = "hello\nworld"
raw_string = r"hello\nworld"
print(f"Chaîne littérale : {string}")
print(f"Chaîne brute : {raw_string}")
```

```
Chaîne littérale : hello
world
Chaîne brute : hello\nworld
```

Chaîne de caractère brute

```
import re


pattern_lit = "\\\\"
pattern_brut = r"\\"
test_lit = "chaine\nono"
test_brut = r"chaine\nono"
for pattern in [pattern_lit, pattern_brut]:
    for test in [test_lit, test_brut]:
        if re.search(pattern, test):
            print("capturé")
        else:
            print("manqué")
```

```
manqué
capturé
manqué
capturé
```

Compilation de pattern — Flag `re.IGNORECASE` (`re.I`)

Rendre insensible à la casse

```
import re


test = "0ab12Ab3491019AB"
pattern = re.compile(r"\d+[ab]")
print(pattern.findall(test))
pattern = re.compile(r"\d+[ab]\D*", re.IGNORECASE)
print(pattern.findall(test))
```

```
['0a']
['0ab', '12Ab', '3491019AB']
```

Compilation de pattern — Flag `re.ASCII` (`re.A`)

Forcer `\w` à fonctionner en mode ASCII

```
import re


test = "Élève très studieux"
pattern = re.compile(r"\w+")
print(pattern.findall(test))
pattern = re.compile(r"\w+", re.ASCII)
print(pattern.findall(test))
```

```
['Élève', 'très', 'studieux']
['l', 've', 'tr', 's', 'studieux']
```

Compilation de pattern — Flag `re.VERBOSE` (`re.X`)

Permet l'écriture d'expressions régulières « lisibles »

```
import re


charref = re.compile(
    r"""
        &[#]                # Début d'une référence d'entité numérique
        (
              0[0-7]+       # Forme octale
            | [0-9]+        # Forme décimale
            | x[0-9a-fA-F]+  # Forme hexadécimale
        )
        ;                   # Point-virgule final
    """,
    re.VERBOSE,
)

charref = re.compile("&#(0[0-7]+|[0-9]+|x[0-9a-fA-F]+);")
```

Compilation de pattern — Autres flags

`re.DOTALL`	Le caractère spécial `.` capture aussi les retours chariot
`re.LOCALE`	Utilisation des définitions de la locale courante pour les captures
`re.MULTILINE`	`^` et `$` matchent les début et fin de ligne, et non de chaîne

Groupe de capture

```python
import re


pattern = re.compile(r"([a-zA-Z]+)(\d+)g*")
r = pattern.finditer("A361224g B4012_w44g")
for res in r:
    print(res.group(0), res.group(1), res.group(2))
```

```
A361224g A 361224
B4012 B 4012
w44g w 44
```

Répétition de groupe

```python
import re


pattern = re.compile(r"(b.d)+")
r = pattern.finditer("abcdbzdbydbhdefgh")
for res in r:
    print(res.group(0), res.group(1), res.span())
```

```
bcdbzdbydbhd bhd (1, 13)
```

Remplacements

```
import re


p = re.compile("(blue|red|green)")
print(p.sub("colour", "blue socks and red shoes"))
print(p.sub("colour", "blue socks and red shoes", count=1))
```

```
colour socks and colour shoes
colour socks and red shoes
```

Remplacements avec groupes

Réutilisation des groupes de capture dans la chaîne de remplacement :

```
import re


test = '{"first_name":"James","last_name":"Bond"}'
p = re.compile(r'^\{"(\S+)":"(\S+)","(\S+)":"(\S+)"\}$')
print(p.sub(r"Person(\1='\2', \3='\4')", test))
```

```
Person(first_name='James', last_name='Bond')
```

Tokenisation : Split

```python
import re


test = "J'aime les expressions régulières"
p = re.compile(r"\W+")
print(p.split(test))
maxsplit = 2
print(p.split(test, maxsplit))
```

```
['J', 'aime', 'les', 'expressions', 'régulières']
['J', 'aime', 'les expressions régulières']
```

Pour aller plus loin

- Guide sur les regex
- Documentation officielle

Avez-vous des questions ?

Travaux pratiques

Instructions

Fonctionnalités d’administration système

Bases de données

Introduction

Les bases de données relationnelles répondent à un besoin :

- De structuration des données
- De découplage de la représentation physique et logiques des données
- De transactionnalité

Concepts clefs

- Les données sont représentées comme un ensemble de tables
- Chaque table est un tableau à deux dimensions
- Chaque point de données est représenté par un nuplet d'attributs
- Des liens entre les tables permettent d'établir des relations entre entités

Table 1 – Personnes

Id	Âge	Poids	Taille
1	52	75	178
2	34	76	165
3	18	60	170

Table 2 – Activités

Id	Type	Durée	IdPersonne
1	Vélo	23	1
2	Nage	46	1
3	Nage	18	3

Autres concepts importants

- Chaque enregistrement a une clef primaire qui l'identifie sans duplicat
- Possibilité de travail transactionnel : toutes les modifications sont effectuées atomiquement, ou aucune n'est effectuée

Relations possibles entre entités

On distingue les relations principalement par leur cardinalité :

- 1 à 1
- 1 à n
- n à 1
- n à n

Avec dans certaines libraires des nuances, par exemple la suppression en cascade ou non.

Langage SQL

Le langage SQL (pour *Structured Query Language*) permet de manipuler les données dans une base de données relationnelle.

Site d'explication du SQL.

Implémentation

Pour intéragir avec des bases de données, une librairie : SQLAlchemy.

Polyvalente, simple pour les cas simples, configurable pour les cas complexes.

Connexion à une base de données

Avec une URI. L'URI contient :

- Le type de base de données
- Le driver
- L'utilisateur
- Le mot de passe
- L'adresse du serveur
- Le nom de la base de données

L'option `echo` permet d'activer les sorties de débug.

```
from sqlalchemy import create_engine

engine = create_engine(
    "postgresql+psycopg2://user:password@host/database",
    echo=True)
```

Lecture de données

```python
from sqlalchemy import select, text

# Style textuel
with engine.begin() as conn:
    result = conn.execute(text("SELECT x, y FROM some_table"))
    for row in result:
        print(f"x: {row.x}  y: {row.y}")

# Utilisation des méthodes de SQLAlchemy Core
with engine.begin() as conn:
    result = conn.execute(select(some_table.c.x, some_table.c.y))
```

Écriture de données

```python
from sqlalchemy import insert, text

# Style textuel
with engine.begin() as conn:
    conn.execute(
        text("INSERT INTO some_table (x, y) VALUES (:x, :y)"),
        [{"x": 6, "y": 8}, {"x": 9, "y": 10}],
    )

# Utilisation des méthodes de SQLAlchemy Core
with engine.begin() as conn:
    conn.execute(insert(some_table),
                 [{"x": 6, "y": 8}, {"x": 9, "y": 10}])
```

ORM

En plus d'être un moteur SQL, SQLAlchemy propose un ORM :

- Modèle décrit par des classes Python
- Objets synchronisés entre la base de données et le code Python

Création de tables

```
from sqlalchemy.orm import declarative_base

Base = declarative_base()


class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    name = Column(String(30))
    fullname = Column(String)


Base.metadata.create_all(engine)
```

Insertion avec l'ORM

```
from sqlalchemy.orm import Session

with Session(engine) as session:
    hugo = User(name="hugo", fullname="Hugo Mougard")
    john = User(name="john", fullname="John Doe")

    session.add_all([hugo, john])

    session.commit()
```

Sélection avec l'ORM

```python
from sqlalchemy import select

user = session.scalars(select(User).where(User.name == "hugo")).first()
```

Compatibilité

SQLAlchemy est compatible avec toutes les grandes BDDs.

Seulement un sous-ensemble de fonctionnalités est exposé pour certaines BDDs.

Gestion des migrations

Pour gérer l'évolution d'une base de données, SQLAlchemy a un projet dédié :

Alembic

Pour aller plus loin

Cours SQLAlchemy

Avez-vous des questions ?

Travaux pratiques

1. Instructions
2. Instructions

Éléments avancés d’administration système

Éléments avancés d'administration système

Requêtes HTTP et scraping

Requêtes HTTP

`requests` permet de manière très simple :

- l'envoi de requêtes HTTP
- la gestion de sessions avec cookies
- le téléchargement & téléversement de fichiers

Et plus encore : documentation officielle

Exemple de code

```
>>> import requests
>>> response = requests.get("https://dummyjson.com/products/1")
>>> response.raise_for_status()
>>> response.text
'{"id":1,"title":"iPhone 9","description":"An apple mobile which is not…
>>> response.json()
{'id': 1, 'title': 'iPhone 9', 'description': 'An apple mobile which is…
```

Scraping de sites

Pour parcourir des sites et récupérer du contenu ciblé, préférer `scrapy` : notebook de démonstration & documentation officielle.

Avez-vous des questions ?

Éléments avancés d'administration système

Envoi de mails

Introduction

Pour l'envoi de mails, on peut passer par la bibliothèque standard ou des bibliothèques tierces :

- Bibliothèque standard : `smtlib`, `ssl` & `email`
- Exemple de bibliothèque tierce : `sendgrid`

Intérêt des bibliothèques tierces : principalement gestion de campagnes mails.

Exemple de code

```
>>> import sendgrid
>>> sendgrid_email = sendgrid.helpers.mail.Mail(
...     from_email="bonjour@exemple.net",
...     to_emails=["dupont@autre-exemple.com", "contact@abc.fr"],
...     subject="Mail envoyé depuis sendgrid",
...     plain_text_content="Voici le corps de mon mail",
... )
...
>>> client = sendgrid.SendGridAPIClient("SG.J…")
>>> client.send(sendgrid_email)
```

Avez-vous des questions ?

Éléments avancés d'administration système

Gestion de serveurs

Introduction

Python offre des possibilités intéressantes pour la gestion de serveurs :

- scripts d'administration locaux
- plugins ansible
- scripts Kubernetes
- scripts Terraform
- …

À plus bas niveau, `fabric` permet d'exécuter des commandes sur des serveurs distants.

Exemple de code `fabric`

```
>>> from fabric import SerialGroup
>>> def disk_free(c):
...     uname = c.run('uname -s', hide=True)
...     if 'Linux' in uname.stdout:
...         command = "df -h / | tail -n1 | awk '{print $5}'"
...         return c.run(command, hide=True).stdout.strip()
...     err = "No idea how to get disk space on {}!".format(uname)
...     raise Exit(err)
...
>>> for cxn in SerialGroup('web1', 'web2', 'db1'):
...     print("{}: {}".format(cxn, disk_free(cxn)))
<Connection host=web1>: 33%
<Connection host=web2>: 17%
<Connection host=db1>: 2%
```

Avez-vous des questions ?

Travaux pratiques

Instructions

Hugo Mougard
hugo@mougard.fr