Python

Séverine Affeldt

October 4, 2018

Abstract

Ce cahier propose une série d'exercices de programmation afin de se familiariser avec le langage Python.

1 Variables et typage

1.1 Nommage

Voici ci-dessous quelques exemples de nommage. Pensez toujours à choisir une nom adéquat, qui permet à une relecteur de comprendre rapidement ce à quoi est destinée cette variable.

```
myvariable = 1
my_variable = 100
myvariable == my_variable
my_variable75 = 75
my_variable > my_variable75
75_myvariable_error = 8
```

On utilise '==' pour tester l'égalité. On utilise '>' pour tester l'inégalité (il existe beaucoup d'autres opérateurs!). Le retour est un booléen. Par convention, on utilise des minuscules pour les noms de variables. Le '_' est également possible. Attention: un nom de variable ne peut pas commencer par un chiffre.

1.2 Typage

Vous pouvez utiliser `type()` pour connaitre le type des variables.

```
type(myvariable)
type('spam')
my_list = [1, 2, 3]
type(my_list)
```

Attention, Python a un typage dynamique. Lors de la programmation objet, l'utilisation de `isinstance` peut se révéler aussi très utile.

1.3 Typage dynamique

On définit la fonction `somme` ci-dessous. En vous aidant des appels suivants, expliquez ce que fait la fonction `somme`.

```
def somme(*largs):
    "retourne la somme de tous ses arguments"
    if not largs:
        return 0
    result = largs[0]
    for i in range(1, len(largs)):
        result += largs[i]
    return result
```

```
somme(12, 14, 300)
```

```
liste1 = ['a', 'b', 'c']
liste2 = [0, 20, 30]
liste3 = ['spam', 'eggs']
somme(liste1, liste2, liste3)
```

```
somme('abc', 'def')
```

Le typage est *dynamique*, car la vérification du type se fait à l'exécution du programme.

Type	Fonction
Entier	`int`
Flottant	`float`
Complexe	`complex`
Chaîne	`str`

Table 1: Conversion de type

1.4 Les types numériques

Essayez les différentes opérations fournies ci-dessous:

```
20*60
(20+2)*(4+8)
48/5                    # division naturelle
48//5                   # quotient
48%5                    # reste
2**10                   # puissance
my_variable = 5
my_variable*10
```

1.5 Les chaînes de caractères

On peut définir et afficher une chaîne de caractères comme ci-dessous:

```
ma_chaine = 'Python est facile'
print(ma_chaine)
```

Il est aussi possible de convertir les caractères en variable numérique:

```
user_age_str = input('Quel est votre age?')
user_age_num = int(user_age_str)
print(user_age_num)
type(user_age_num)
```

1.6 Affectations et opérations

Il est facile d'incrémenter la valeur d'une variable:

```
ma_variable = 10
print(ma_variable)
ma_variable += 2
print(ma_variable)
```

Un syntaxe similaire est possible avec d'autres opérateurs:

```
ma_variable -= 4
print('ma_variable =', ma_variable)
ma_variable *= 2
print('ma_variable =', ma_variable)
ma_variable /= 2
print('ma_variable =', ma_variable)
```

Cette syntaxe peut aussi s'appliquer à d'autres types d'objet:

```
ma_list = [0, 3, 5]
print('ma_list', ma_list)

ma_list += ['a', 'b']
print('ma_list', ma_list)
```

2 Introduction à la programmation

2.1 Les chaînes de caractères

De nombreuses opérations sont possibles sur les chaînes de caractères. Vous trouverez toutes les informations dans le manuel à l'aide de la commande `help(str)`. Ci-dessous, vous trouverez un aperçu de l'utilisation des commandes les plus fréquemment rencontrées.

Les commandes `split` et `join` vous permettrons de découper ou d'assembler des chaînes de caractères.

```
# Decoupage
'welcome..=..bienvenida..=..youkoso'.split('..=..')

# Assemblage
'..=..'.join(['welcome', 'bienvenida', 'youkoso'])
```

Attention aux cas où le dernier caractère est le séparateur!

```
# Decoupage
'welcome;bienvenida;youkoso;'.split(';')

# Assemblage
';'.join(['welcome', 'bienvenida', 'youkoso', ''])
```

Il est possible de remplacer très facilement des chaînes ou sous-chaînes de caractères avec la fonction `replace`.

```
# Tout remplacer
'welcome;bienvenida;welcome'.replace('welcome', '!!!')

# Remplacer une fois
'welcome;bienvenida;welcome'.replace('welcome', '!!!', 1)
```

Une autre fonction utile est la fonction `strip` qui permet *nettoyer* la début et la fin d'une chaîne de caractère en enlevant les tabulations, les espaces ou les retours à la ligne. On peut également spécifier le caractère à enlever, ce qui peut se révéler intéressant avant d'appeler `split`.

```
# Nettoyage par defaut
print('␣\twelcome;bienvenida;welcome␣\n')
'␣\twelcome;bienvenida;welcome␣\n'.strip()

# Nettoyage specifique
'welcome;bienvenida;welcome'.strip(';').split(';')
```

Il est parfois nécessaire de rechercher l'indice d'une sous-chaîne de caractère. De nombreuses fonctions permettent de faire ce travail.

```
# Trouver l'indice du debut de la premiere occurence
'bienvenida;youkoso;bienvenida'.find('veni')
'bienvenida;youkoso;bienvenida'.find('xyz')

# ... en partant de la fin
'bienvenida;youkoso;bienvenida'.rfind('veni')
'bienvenida;youkoso;bienvenida'.rfind('xyz')
'bienvenida;youkoso;bienvenida'[23]
'bienvenida;youkoso;bienvenida'[23:27]
```

On peut aussi utiliser la fonction `index` qui permet de lever une exception dans le cas où la chaîne recherchée n'existe pas.

```
# Trouver l'indice du debut de la premiere occurence
'bienvenida;youkoso;bienvenida'.index('veni')
'bienvenida;youkoso;bienvenida'.index('xyz')

# Comment attraper l'exception
try:
        'bienvenida;youkoso;bienvenida'.index('xyz')
except Exception as e:
        print('Cannot␣find␣the␣string', type(e), e)
```

Vous pouvez aussi plus simplement demander si la chaîne existe avec `in` ou combien de fois elle apparaît avec `count`. Essayez également les fonctions `startswith` et `endswith`!

```
# La chaine existe-t-elle?
'veni' in 'bienvenida;youkoso;bienvenida'
'bienvenida;youkoso;bienvenida'.count('veni')
```

Enfin, il est possible de modifier facilement la casse des caractères.

```
'bienvenida␣YOUKOSO'.upper()
'bienvenida␣YOUKOSO'.lower()
'bienvenida␣YOUKOSO'.swapcase()
'bienvenida␣YOUKOSO'.capitalize()
'bienvenida␣YOUKOSO'.title()
```

2.2 L'affichage des chaînes de caractères

La fonction `print` est très utilisée. Elle permet d'insérer des espaces entre les chaînes à afficher et un saut de ligne (qui peut être supprimé si nécessaire).

```
print("ma", "premiere", "ligne")
print("ma", "deuxieme", "ligne", end = ' ')
print("sans saut !")
```

On peut également afficher des objets avec `print`.

```
# Affichage d'un module
import math

print("Mon objet math est", math)

# Affichage d'une instance de classe
# --> la classe
class Etudiant:
        pass

# --> l'instance
un_etudiant = Etudiant()

# --> l'affichage
print(un_etudiant)
```

Pour des affichages complexes, on préfère utiliser le formatage. Pour cela, `python3.6` propose les `f-string`. Les accolades des `f-string` englobent du code python.

```
pays, capitale, n_hab_M = "Japon", "Tokyo", 126
f"{pays} ({capitale}, {n_hab_M} millions)"
f"{pays} ({capitale}, {n_hab_M+1} millions)"
```

Les versions antérieures de python proposent le formatage via la fonction `format`.

```
pays, capitale, n_hab_M = "Japon", "Tokyo", 126
"{} ({}, {} millions)".format(pays, capitale, n_hab_M)
"{1} ({0}, {2} millions)".format(capitale, pays, n_hab_M)
"{py} ({cp}, {hb} millions)".format(py=pays, cp=capitale, hb=n_hab_M)
```

Le grand intérêt du formatage réside dans la gestion de l'affichage des nombres.

```
# Avec f-string (python3.6)
from math import pi
f"pi, 2 chiffres apres la virgule {pi:.2f}"
```

```
# Avec format (python3.5)
from math import pi
"pi, 2 chiffres apres la virgule {flottant:.2f}".format(flottant=pi)
```

On peut également forcer la largeur des nombres.

```
# Avec f-string (python3.6)
n_hab_M = 127
f"{n_hab_M:04d}"
```

On peut afficher des données en colonnes de largeur fixe. On indique avec <, ^ et > l'affichage à gauche, au centre, ou à droite d'une zone de largeur fixe.

```
monde = [
 ('Japon', 'Tokyo', 127),
 ('France', 'Paris', 67),
 ('Norvege', 'Oslo', 5 ),
]

for pays, capitale, n_hab_M in monde:
    print(f"{capitale:<10} -- {pays:^12} -- {n_hab_M:>8} millions")
```

2.3 Dialoguer avec l'utilisateur

Il est parfois nécessaire de solliciter l'utilisateur pour obtenir des informations. On peut utiliser pour cela `input`, qui renvoie une chaîne de caractères. Attention aux conversion!

```
nom_pays = input("Entrez le nom du pays : ")
print(f"Vous avez saisi '{nom_pays}'")
int(input("Nombre d'habitant en millions ?")) + 1
```

En générale, on utilise plutôt le module `argparse` pour passer des arguments au programme via une ligne de commande. **Recherchez des informations sur ce module et créez un script python qui prend en ligne de commande le nom d'un pays, sa capitale, son nombre d'habitants en millions, et qui affiche ces informations.**

2.4 Les expressions régulières

...TBD...

2.5 Manipuler une séquence

On peut faire du *slicing* sur les chaînes de caractères. Il faut retenir que:

- les indices commencent toujours à 0
- le premier indice est inclus
- le dernier indice est exclu

```
ma_chaine = "bienvenida;youkoso;welcome"
print(ma_chaine)

ma_chaine[2:6]
ma_chaine[0:3] + ma_chaine[3:7] == ma_chaine[0:7]
ma_chaine[:6]
ma_chaine[24:]
ma_chaine[:]

# Slicing en partant de la fin
ma_chaine[3:-3]
ma_chaine[-3:]

# Slicing avec pas
ma_chaine[3:-3:2]
ma_chaine[3:-4:2]

# Slicing avec pas negatif
ma_chaine[-3:3]
ma_chaine[-3:3:-2]
```

On peut appliquer ces *slices* sur des objets `list`. **Définissez `ma_liste = [0, 2, 4, 8, 16, 32, 64, 128]` et appliquez les slicing précédents.**

2.6 Les listes

Nous allons voir dans cette sous-section différentes fonctions qui permettent de manipuler les listes.

```
ma_liste = [0,1,2,3]
print("Ma liste ", ma_liste)
```

On peut ajouter un élement (`append`) ou plusieurs éléments (`extend`) à une liste.

```
# Un element
ma_liste.append('ap')
print("Ma liste ", ma_liste)     # on a modifie ma_liste

# Plusieurs elements
ma_liste.extend(['ex1', 'ex2'])
print("Ma liste ", ma_liste)     # on a modifie ma_liste
```

L'opérateur `+` permet également d'ajouter des éléments, mais il ne modifie pas l'objet. Il en crée un nouveau.

```
# Un element
ma_liste2 = [4,5,6]
print("Ma liste2 ", ma_liste2)

# Plusieurs elements
ma_liste3 = ma_liste + ma_liste2
print("Ma liste ", ma_liste)     # aucune modification de ma_liste
print("Ma liste2 ", ma_liste2)   # aucune modification de ma_liste2
print("Ma liste3 ", ma_liste3)   # creation de ma_liste3
```

On peut également insérer des éléments(`insert`), enlever des éléments (`remove`, `pop`) ou renverser la liste (`reverse`).

```
ma_liste = [0,1,2,3]
ma_liste.insert(2, '1 bis')
print('Ma liste', ma_liste)

ma_liste[5:5] = ['3 bis']
print('Ma liste', ma_liste)
```

```
ma_liste.remove(3)       # Remove the element '3'
print('Ma liste', ma_liste)
```

```
popped = ma_liste.pop(0) # On precise le premier indice
print('popped', popped, 'ma liste', ma_liste)

popped = ma_liste.pop() # Par defaut c'est le dernier indice
print('popped', popped, 'ma liste', ma_liste)
```

```
ma_liste.reverse()
print('Ma liste', ma_liste # ma_liste est modifie
```

La fonction `sort` permet de trier une liste. Attention, la liste sur laquelle est fait l'appel est modifiée. Si vous ne souhaitez pas modifier votre liste initiale, il faut utiliser `sorted`.

```
# Tri croissant
ma_liste = [8, 7, 4, 3, 2, 9, 1, 5, 6]
print('avant tri', ma_liste)
ma_liste.sort()
print('apres tri', ma_liste)

# Tri decroissant
ma_liste = [8, 7, 4, 3, 2, 9, 1, 5, 6]
print('avant tri', ma_liste)
ma_liste.sort(reverse=True)
print('apres tri decroissant', ma_liste)

# Pas de modification de ma_liste avec sorted
sorted(ma_liste)
print('apres tri croissant via sorted', ma_liste)

# Avec les strings
ma_liste_str = ['spam', 'egg', 'bacon', 'beef']
ma_liste_str.sort()
print('apres tri str', ma_liste_str)
```

Soyez prudents quand il s'agit de classement de strings. En effet, il faut comparer des strings avec la même casse et correctement nettoyés (pas d'espace en début par exemple).

2.7 Les tests `if`

Il est préférable de respecter une indentation à 4 espaces (8 en cas d'imbrication etc...). Attention, python est sensible aux espaces et indentations! Veillez notamment à vérifier la bonne conversion de votre `tab` en espaces.

```
# Code accepte
if 's' in 'Paris Descartes':
        print('Oui')
else:
        print('Non')
```

```
# Code rejete
if 's' in 'Paris Descartes':
        print('Oui')
        else:
                print('Non')
```

2.8 Les fonctions

On définit une fonction à l'aide du mot-clef `def`. La fonction ci-dessous renvoie le carré d'un nombre.

```
def carre(n):
        "Cette fonction donne le carre d'un nombre n"
        return n*n

# Appel de la fonction
mon_carre = carre(2)
print(mon_carre)
```

Le mot-clef `return` permet de mettre fin à la fonction. Si un élément est spécifié sur la même ligne, il est retourné par la fonction. Sinon, la fonction retourne `None`.
La fonction ci-dessous retourne

- True si le caractere car est dans 'Descartes' avec la meme casse
- False si le caractere car est dans 'Descartes' avec une casse differente
- None, sinon

```
def dummy_print(car):
    ref_str = 'Descartes'
    if car in ref_str:
        return True
    elif car.lower() in ref_str.lower():
        return False
    else:
        return None

# Appel de la fonction
dummy_print('d')
dummy_print('D')
dummy_print('z')
```

3 Notions avancées de programmation python

3.1 Les fichiers

Il est recommandé d'ouvrir le fichier avec `with` qui garantit la fermeture du fichier. Faites attention à l'encodage!

```
with open("/home/saffeldt/foo.txt", "w", encoding='utf-8') as sortie:
    for i in range(2):
        sortie.write(f"{i}\n")
```

Vous avez plusieurs modes possible d'ouverture:

- 'r' ouverture en lecture
- 'w' ouverture en écriture (attention à l'écrasement de fichier)
- 'a' ouverture en écriture pour ajout (append)
- '+' ouverture en lecture et écriture
- 'b' ouverture en mode binaire

```
with open("/home/saffeldt/foo.txt", "a", encoding='utf-8') as sortie:
    for i in range(4):
        sortie.write(f"{i}\n")
```

Vous pouvez visualiser le contenu du fichier dans le terminal.

```
# maintenant on regarde ce que contient le fichier
with open("/home/saffeldt/foo.txt", encoding='utf-8') as entree:
    for line in entree:
        # line contient deja un retour a la ligne
        print(line, end='')
```

Le module `pathlib` fournit une interface orientée objet qui permet de parcourir l'arborescence des fichiers. Ce module propose notamment une classe appelée `Path`.

```
from pathlib import Path

nom_fichier = "fichier-test"
path_fichier = Path(nom_fichier)
# Ecrire path_fichier. et appuyer deux foix sur tab pour voir les fonctions

# Existence
path_fichier.exists()
# Ouverture en ecriture
with open(nom_fichier, 'w', encoding='utf-8') as output:
    output.write('0123456789\n')

# Existence
path_fichier.exists()
```

On peut lire de nombreuses informations à partir de l'objet instancié.

```
# la taille du fichier en octets est de 11
# soit 10 caracteres et un saut de ligne
path_fichier.stat().st_size

# la date de derniere modification est le nombre
# de secondes depuis le 1er Janvier 1970
mtime = path_fichier.stat().st_mtime
mtime

# que l'on peut rendre plus lisible comme ceci
# en anticipant sur le module datetime
from datetime import datetime
mtime_datetime = datetime.fromtimestamp(mtime)
mtime_datetime
```

Le destruction d'un fichier peut se faire via la méthode `unlink`.

```
# Detruire le fichier seulement s'il existe
try:
    path_fichier.unlink()
except FileNotFoundError:
    print("no need to remove")

path_fichier.exists()
path_fichier.name
```

On peut aussi utiliser une instance de `Path` pour rechercher des fichiers dans un dossier.

```
# Recherche des fichiers .json dans le dossier data
dirpath = Path('./data/')

# tous les fichiers *.json dans le repertoire data/
for json in dirpath.glob("*.json"):
    print(json)
```

3.2 Les formats de fichiers

Le format JSON (JavaScript Object Notation) est très populaire. C'est un encodage qui se prête bien aux types de base des langages modernes tels que Python, Ruby ou JavaScript.

```
import json
# Objet defini avec des types de base
data = [
    [1, 2, 'a', [3.23, 4.32], {'eric': 32, 'jean': 43}],
    # un tuple
    (1, 2, 3),
]

# Sauvegarde dans un fichier JSON
with open("s1.json", "w", encoding='utf-8') as json_output:
        json.dump(data, json_output)

# Lecture du fichier JSON
with open("s1.json", encoding='utf-8') as json_input:
        data2 = json.load(json_input)
```

JSON est un format très pratique mais il y a quelques limitations comme la conversion inévitable des tuples en listes...
Parmi les autres formats, on rencontre aussi:

- .csv: peut rendre service à l'occasion...
- pickle: format dédié Python, très proche de JSON et utile pour la sauvegarde d'objets en cours de programme
- .xml: plus flexible que JSON mais plus complexe à mettre en place

3.3 Les tuples

Les tuples sont des objets *non mutables*. C'est-à-dire qu'on ne peut pas changer les valeurs d'un tulple après sa création. Par opposition, une liste est mutable. Une liste ne peut donc pas être utiliser comme une clef de dictionnaire. C'est possible avec un tuple.

```
# Plusieurs facon de declarer un tuple

# sans parentheses ni virgule terminale
tuple1 = 1, 2
# avec parentheses
tuple2 = (1, 2)
# avec virgule terminale
tuple3 = 1, 2,
# avec parentheses et virgule
tuple4 = (1, 2,)

# Tuple avec un element
simple3 = 1,
simple4 = (1,)

# Tuple qui comporte des listes
# Notez la derniere virgule...
mon_tuple = ([1, 2, 3],
             [4, 5, 6],
             [7, 8, 9],
            )
mon_tuple[0]
mon_tuple[0]='a'
mon_tuple[0][1]='a'

# Un tuple a partir d'une liste
mon_tuple = tuple([1,2,3])
```

Il est possible de faire des opérations avec des tuples.

```
tuple1 = (1, 2,)
tuple2 = (3, 4,)
print('addition', tuple1 + tuple2)

tuple1 = (1, 2,)
tuple1 += (3, 4,)
print('apres_ajout', tuple1)
```

3.4 Sequence unpacking

En python, on peut initialiser en même temps plusieurs variables à partir d'une seule.

```
# A partir d'un tuple
mon_tuple = (1, 2)
gauche, droite = mon_tuple
gauche
droite

# A partir d'une liste
ma_liste = [1,2,3]
un, deux, trois = ma_liste
```

Ce type d'affection simplifie certaines opérations, telle que l'échange de variables.

```
a = 1
b = 2
a, b = b, a
print('a', a, 'b', b)
```

Une version *extended* du unpacking permet d'affecter plusieurs valeurs à une seule variable.

```
reference = [1, 2, 3, 4, 5]
a, *b, c = reference
print("a={} b={} c={}".format(a, b, c))

# Si les valeurs du milieu de nous interesse pas
a, *_, c = reference
print("a={} c={}".format(a, c))
```

On peut aussi faire du unpacking en profondeur

```
structure = ['abc', [(1, 2), ([3], 4)], 5]
(a, (b, ([trois], c)), d) = structure
print('trois', trois)
```

3.5 Les boucles `for` à plusieurs variables

La boucle suivante itère sur une seule liste mais agit sur plusieurs variables.

```
entrees = [(1, 2), (3, 4), (5, 6)]
for a, b in entrees:
    print("a={} b={}".format(a,b))
```

Le script ci-dessous fournit une exemple d'exploitation de plusieurs variables par une boucle après l'utilisation de `zip` pour associer deux listes.

```
pays = ["Japon", "France", "Norvege"]
particule = ["au", "en", "en"]
populations = [127, 67, 5]

# Regardez ce que produit zip
list(zip(pays, particule, populations))

# Boucle
for py, part, population in zip(pays, particule, populations):
    print(population, "millions d'habitants", part, py)
```

Attention au résultat tronqué avec `zip` si les listes ne sont pas de la même taille.

L'énumération est une autre façon d'itérer.

```
for i, py in enumerate(pays):
    print(i, py)
```

3.6 Les dictionnaires

Un dictionnaire comporte des clefs et des valeurs.

```
# (1) Creation d'un dictionnaire
france = {'Nord': 59, 'Creuse': 23, 'Loire': 42}
print(france)

# (2) Creation d'un dictionnaire
france = dict([('Nord', 59), ('Creuse', 23), ('Loire', 42)])
print(france)

# (3) Creation d'un dictionnaire
france = dict(Nord = 59, Creuse = 23, Loire = 42)
print(france)
```

On peut accéder aux valeurs si la clef existe, les modifier et les supprimer.

```
print('Le numero de departement du Nord est', france['Nord'])

# Existance d'une clef
print('Nord' in france)
print('XYZ' in france)

# La clef n'existe peut-etre pas, on donne une valeur par defaut
print('Le numero de departement du Nord est', france.get('Nord', 0))
print('Le numero de departement de XYZ est', france.get('XYZ', 0))

# On peut modifier une valeur et ajouter une entree
france['Nord'] = 62
```

```
print(france)
france['Nord'] = 59
france['Pas-de-calais'] = 62
print(france)

# Suppression d'une entree
del france['Pas-de-calais']
print(france)
```

Deux méthodes permettent d'obtenir séparément les clefs et les valeurs: `keys()` et `values()`.

```
for clef in france.keys():
        print(clef)
for valeur in france.values():
        print(valeur)
```

On peut aussi parcourir un dictionnaire et lire les clefs et valeurs en même temps.

```
for departement, numero in france.items():
    print('Dept {} et num {}'.format(departement, numero))
```

Attention, un dictionnaire n'est jamais ordonné. Pour avoir un dictionnaire ordonné, il faut utiliser la classe `OrderedDict` du module `collections`.

3.7 Les exceptions

On peut attraper des exceptions avec `try`. Ce mot-clef peut-être associé à `else` et `finally`. `else` est exécuté si aucune exception n'est attrapée. `finally` est executée quoi qu'il arrive.

```
# une fonction qui fait des choses apres un return
def return_with_finally(number):
    try:
        return 1/number
    except ZeroDivisionError as e:
        print("OOPS, {}, {}".format(type(e), e))
        return("zero-divide")
    finally:
        print("on passe ici meme si on a vu un return")

# Appels
# sans exception
return_with_finally(1)

# avec exception
return_with_finally(0)
```

4 Conception de classes

Une classe est un type spécifique définie par le programme.

Supposons qu'on souhaite créer une classe qui comport une matrice 2x2 et une fonction de calcul du déterminant.

```
class Matrix2:
    """Une deuxieme implementation, tout aussi
    sommaire, mais differente, de matrice carree 2x2"""

    def __init__(self, a11, a12, a21, a22):
        "construit une matrice a partir des 4 coefficients"
        # on decide d'utiliser un tuple plutot que de ranger
        # les coefficients individuellement
        self.a = (a11, a12, a21, a22)

    def determinant(self):
        "le determinant de la matrice"
        return self.a[0] * self.a[3] - self.a[1] * self.a[2]

    def __repr__(self):
        "comment presenter une matrice dans un print()"
        return "<<mat-2x2 {}>>".format(self.a)

help(Matrix2)
```

On peut alors créer une instance de la classe et appeler la méthode qui calcule le déterminant.

```
matrice = Matrix2(1, 2, 2, 1)
print(matrice)

matrice.determinant()
```

Attention, en python il n'y pas d'équivalent à `Public`, `Private` et `Protected`.

En revanche, il y a bien une notion d'héritage.

```
# Une classe mere
class Fleur:
    def implicite(self):
        print('Fleur.implicite')

    def redefinie(self):
        print('Fleur.redefinie')

    def modifiee(self):
        print('Fleur.modifiee')

# Une classe fille
class Rose(Fleur):
    # on ne definit pas implicite
    # on redefinit completement redefinie
    def redefinie(self):
        print('Rose.redefinie')

    # on change un peu le comportement de modifiee
    def modifiee(self):
        Fleur.modifiee(self)
        print('Rose.modifiee apres Fleur')
```

On peut créer une instance de Rose, et appeler l'instance sur 3 méthodes.

```
# fille est une instance de Rose
fille = Rose()
fille.implicite()

fille.redefinie()

fille.modifiee()
```

5 Outils Python pour la data science

5.1 numpy

5.1.1 Les fonctions basiques de `numpy`

Le module `numpy` fournit le type `array`.

On peut facilement créer des tableaux 1D.

```
import numpy as np
array = np.array([12, 25, 32, 55])
array
```

On peut facilement créer des suites de nombre et les placer dans une matrice.

```
numpy_range = np.arange(10)
numpy_range

numpy_range_f = np.arange(1.0, 2.0, 0.1)
numpy_range_f

X = np.linspace(0., 10., 50)
X
```

On peut tracer la courbe des valeurs.

```
import matplotlib.pyplot as plt
Y = np.cos(X)
plt.plot(X, Y);
```

On accède au type du tableau via `.dtype`.

attribut	signification	exemple
shape	tuple des dimensions	(3,5,7)
ndim	nombre dimensions	3
size	nombre d'éléments	3 * 5 * 7
dtype	type de chaque élément	np.float64
itemsize	taille en octets d'un élément	8

Table 2: Conversion de type

```
import numpy as np
a = np.array([1, 2, 4, 8])
a.dtype
```

Il est possible de choisir explicitement le type.

```
c2 = np.array([1, 2, 4, 8], dtype=np.complex64)
c2.dtype
```

On peut obtenir la forme du tableau avec l'attribut `shape`.

```
d2 = np.array([[11, 12, 13], [21, 22, 23]])
d2
d2.shape          # on obtient lignes x colonnes
```

Un tableau est une vue, on peut donc changer de forme facilement.

```
# l'argument qu'on passe a reshape est le tuple
# qui decrit la nouvelle shape
v2 = d2.reshape((3, 2))
v2
# Attention aux dimension incompatibles
try:
    d2.reshape((3, 4))
except Exception as e:
    print(f"OOPS {type(e)} {e}")
```

Il existe différents attributs lies a la forme

```
# le nombre de dimensions
d2.ndim
# le nombre de cellules
d2.size
# vrai pour tous les tableaux
len(d2.shape) == d2.ndim
```

On peut également créer des tableaux de dimension supérieures à 2.

```
shape = (2, 3, 4)
size = reduce(mul, shape)

# vous vous souvenez de arange
data = np.arange(size)
d3 = data.reshape(shape)
d3                # n1 x n2 x lignes x colonnes
```

Il es toujours possible de manipuler les dimensions en ajoutant un 1 sans changer la taille du tableau.

```
d2.shape
d2
d2.reshape(2, 1, 3)
d2.reshape(2, 3, 1)
```

Le tableau ci-dessous résume les différents attributs disponibles.
Enfin, la méthode `ravel` permet d'aplatir n'importe quel tableau.

```
d2
d2.ravel()
# il y a d'ailleurs aussi flatten qui fait
# quelque chose de semblable
d2.flatten()
```

5.1.2 Créer des tableaux

Python autorise la création de tableaux vides.

```
my_tab = np.empty( dtype = np.int8 , shape = (1_000 , 1_000))
print(my_tab)
```

Attention: vous n'avez pas créé une matrice de 0! Pour cela, il faut utiliser les commandes suivantes:

```
# Matrice de 0
my_tab_zeros = np.zeros( dtype = np.int8 , shape = (1000, 100))
print(my_tab_zeros)

# De meme on peut crer une matrice de 1
my_tab_ones = np.ones( dtype = np.int8 , shape = (1000, 100))
print(my_tab_ones)

# ou bien une matrice de 4
my_tab_fours = 4 * np.ones( dtype = np.int8 , shape = (1000, 100))
print(my_tab_fours)
```

La fonction `indices` permet de créer très facilement des matrices à plusieurs dimensions.

```
# On souhaite construire une table telle que
# my_tab[i, j] = 200*i + 2*j +50

# On commence par creer des tableaux d'indices
ix, iy = np.indices((2, 4))
print(ix)
print(iy)

# Puis on rempli la table
my_tab = 200*ix + 2*iy +50
print(my_tab)
```

5.1.3 Le broadcasting

Le *broadcasting* est l'acion de combiner des tableaux/matrices de dimensions différentes mais compatibles via des opérateurs (eg., +, -, *).

```
# On definit une matrice 'a' et un entier 'b'
# ('b' est en fait comme un tableau de dim. (1,))
a = 100 * np.ones((3, 5), dtype = np.int32)
b = 3

# On peut ajouter ces deux variables
c = a + b                # NB: nouvelle allocation...
print(c)
print(c - a)
```

On peut également ajouter un vecteur à chaque ligne de la matrice ou à chaque colonne.

```
# On definit un vecteur 'b'
b = np.arange(1,6)
print(b)

# On ajoute ce vecteur a chaque ligne de 'a'
c = a + b                # NB: nouvelle allocation...
print(c)
print(c - a)

# On redefinit le vecteur 'b'
b = np.arange(1,4).reshape(3, 1)
print(b)

# On ajoute ce vecteur a chaque colonne de 'a'
c = a + b                # NB: nouvelle allocation...
print(c)
print(c - a)
```

On considère à présent deux matrices qui ont a priori des dimensions incompatibles. Le broadcasting va permettre d'*étirer* les dimensions d'origine pour obtenir une dimension commune.

```
tab_col = np.arange(1,4).reshape((3,1))
tab_lg = 100 * np.arange(1,6)
print(tab_col)
print(tab_lg)
```

```
# On ajout ces deux variables
m = tab_col + tab_lg
print(m)
```

Avec le broadcasting, on a en quelque sort transformé `tab_col` de dimension `(3, 1)` en dimension `3, 5`, et `tab_lg` de dimension `(1, 5)` en dimension `3, 5`.
Une dernière exemple de broadcasting est donné ci-dessous,

```
i = np.arange(5)
my_array = i + 10*i.reshape(5,1)
print(my_array)
```

5.1.4 Index et slices

Il est très simple d'accéder aux élements d'une numpy array,

```
# Acceder a une ligne
my_array[1]

# Acceder a un element
my_array[1][2]

# Acceder a un tuple
my_array[1,2]
```

De même, on peut très facilement modifier une array,

```
# Modifier un element
my_array[2][1] = 221
my_array[3,2] += 300
print(my_array)

# Modifier une ligne
my_array[1] = np.arange(100, 105)
print(my_array)

# Modifier une ligne avec broadcasting
my_array[4] = 400
print(my_array)
```

On peut utiliser le slicing pour accéder à une colonne et la modifier,

```
# Extraire la colonne 4
my_array[:, 3]

# Modifier la colonne 4
my_array[:, 3] = range(300, 305)
print(my_array)

# Modifier la colonne 3 puis la 5 avec broadcasting
my_array[:, 2] = 200
print(my_array)
my_array[:, 4] += 400
print(my_array)
```

Il est possible d'extraire plusieurs lignes/colonnes grâce au slicing,

```
# Extraire les lignes/colonnes d'indice 0, 2 et 4
my_array[0::2]
my_array[:,0::2]

# Extraire un bloc
my_array[1:3, 2:5]
```

5.1.5 Opération logique

On peut utiliser `np.all` pour savoir si deux arrays sont identiques,

```
np.all(my_array == my_array)
```

La fonction `any` permet elle de vérifier si au moins un élément est à vrai,

```
np.zeros(5).any()
np.ones(5).any()
```

5.1.6 Algèbre linéaire

Pour faire un produit matriciel en Python, il faut utiliser la méthode `dot` du package `numpy`. Attention, '*' fait un produit terme à terme, qui est différent du produit matriciel.

```
# Ceci n'est pas un produit matriciel!
A = 1 + np.arange(3)
B = 1 + np.arange(3).reshape(3,1)
print(A)
print(B)
print(A*B)

# Ceci est un produit matriciel
np.dot(A, B)

# Ceci est egalement un produit matriciel
A = np.array([[1, 1],
              [2, 2]])
B = np.array([[10, 20],
              [30, 40]])
np.dot(A,B)

# L'operateur @ permet aussi de faire un produit matriciel
A @ B

# dot permet aussi de faire un produit scalaire
A = np.array([1,2,3])
B = np.array([4,5,6])
np.dot(A, B)
```

Il existe deux façon de transposer une matrice,

```
A = np.arange(4).reshape(2,2)

# Avec T
A.T

# Avec transpose
A.transpose()
```

Il est facile d'obtenir la matrice identité, d'obtenir la diagonales d'une matrice, ou de construire une matrice diagonale,

```
# Matrice identite
np.eye(4, dtype=np.int)

# Extraire la diagonale
A = np.arange(9).reshape(3,3)
D = np.diag(A)
print(D)

# Construire une matrice a partir d'une diagonale
A2 = np.diag(D)
print(A2)
```

La fonction `concatenate` permet d'assembler des tableaux,

```
A = np.ones((2,3,4))
B = np.zeros((2,3,2))
np.concatenate((A,B), axis = 2)
```

Le déterminant d'une matrice se calcule simplement avec `np.linalg.det`,

```
A = np.array([[0,1], [-1, 0]])
print(A)
print(np.linalg.det(A))
```

On obtient les vecteurs et valeurs propres d'une matrice avec `np.linalg.eig`

```
A = np.array([[0,1], [1, 0]])
values, vectors = np.linalg.eig(A)
print(values)
print(vectors)
```

5.2 pandas

5.2.1 Introduction aux data frames

`pandas` permet l'indexation de `array` de `numpy` et cette indexation rend l'utilisation des `array` très efficace. Pour la suite ce cette section, on charge un jeu de données issu du module `seaborn`,

```
import seaborn as sns
tips = sns.load_dataset('tips')
type(tips)

# Visualiser les premieres lignes
tips.head()
```

On voit bien un index colonnes (noms) et un index lignes (rangs).

On peut facilement obtenir des statistiques globales sur le jeu de données ou spécifiques à une variable,

```
# Pour la table
tips.describe()

# Pour la variables sexe
tips.groupby('sex').mean()

# Pour la variables day
tips.groupby('day').mean()

# Pour la variable time
tips.groupby('time').mean()
```

5.2.2 Les Series

5.3 matplotlib

5.4 Utiliser `Python` ET `R` ensemble

Un package `R`, `Reticulate`, permet l'utilisation efficace de `Python` dans un environnement `R`. Nous allons voir comment mettre en oeuvre cette association.
A l'aide de `RStudio`, installez le package `reticulate`. Vous pouvez ensuite définir la version par défaut de `Python` que vous souhaitez utiliser,

```
library(reticulate)
use_python("use_python("/usr/bin/python3.5")")  # par exemple...
```

Pour une version de `Python` dans un environnement virtuel,

```
library(reticulate)
use_virtualenv("myenv")
```

Nous allons voir trois façons différentes d'appeler du code `Python` dans une environnement `R`.

5.4.1 Import de modules `Python`

Dans le code suivant, on importe le module `Python os` pour utiliser la fonction `listdir()`,

```
library(reticulate)
os <- import("os")
os$listdir(".")
```

Avec `tab` (après le $), il est possible d'avoir la liste des fonctions `Python` disponibles dans le module importé.

5.4.2 Exécution de scripts `Python`

On crée dans un premier temps le script `Python` ci-dessous et on le sauvegarde sous le nom 'flights.py',

```
import pandas
def read_flights(file):
  flights = pandas.read_csv(file)
  flights = flights[flights['dest'] == "ORD"]
  flights = flights[['carrier', 'dep_delay', 'arr_delay']]
  flights = flights.dropna()
  return flights
```

Dans l'environnement `R`, les commandes ci-dessous permettent d'exécuter le script précédent,

```
# Execution du script Python comportant la fonction de chargement
source_python("/path/to/flights.py")

# Utilisation dans R de la fonction de chargement Python
flights <- read_flights("/path/to/flights.csv")

# Le contenu du fichier est maintenant dans une variable R
head(flights)
summary(flights)

# On manipuler cette variable dans l'environnement R
library(ggplot2)
ggplot(flights, aes(carrier, arr_delay)) + geom_point() + geom_jitter()
```

5.4.3 Travailler de façon interactive avec `Python`

Il est possible d'ouvrir une session `Python` dans une session `R`. Pour cela il suffit d'utiliser la fonction `repl_python()`. L'objet `py` dans `R` contiendra alors vos objets/variables/fonctions `Python`,

```
# Dans une session R, tapez
repl_python()
## vous etes a present dans une session R

# Dans la session R, tapez les lignes suivantes
import pandas
flights = pandas.read_csv("/path/to/flights.csv")
flights = flights[flights['dest'] == "ORD"]
flights = flights[['carrier', 'dep_delay', 'arr_delay']]
flights = flights.dropna()

# Vous pouvez quitter l'environnement Python pour retourner a R
exit

# Vous pouvez acceder aux objet Python avec py$
py$flights
```

IMPORTANT: Vous pouvez accéder aux objets/variables/fonctions R dans Python avec `r.`