Programmation Objet

1-1: Rappels sur le language Python

VCOD 2022/2023

Maxime Raynal

But de ces rappels:

- Se rafraichir la mémoire
- Introduire du vocabulaire
- Faire un tour rapide de quelques fonctionnalités à connaître du langage.

Ce qu'on va couvrir:

- Le Zen de Python
- Variables et assignements
- Priorités des opérateurs
- Les chaînes de caractères
- Les commentaires
- Debugger
- Conditionalités

Ce qu'on va survoler aujourd'hui:

- Le Zen de Python
- Syntaxe et indentations
- Variables et assignements
- Priorités des opérateurs
- Les chaînes de caractères
- Les commentaires
- Gestion d'erreurs et debug
- Conditions
- Fonctions
- Itérations
- Listes
- Dictionnaires
- Tuples
- Ensembles
- Les "*comprehensions*"
- Autres structures de données

Le Zen de Python

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.

- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one– and preferably only one – obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea – let's do more of those!

Syntaxe et indentations

- En Python, on n'est pas obligés de finir une ligne par ';'.
  - Mais on peut.

Syntaxe et indentations

- En Python, on n'est pas obligés de finir une ligne par ';'.
  - Mais on peut.
- **Les indentations font partie du langage !**
  - Immense majorité des langages de programmation -> les indentations ne sont là que pour aider les développeurs à lire.
  - En Python, elles font partie de la sémantique du langage.
  - Remplacent les paires d'accolades de beaucoup d'autres langages (C, Java, PHP, JS …).

Syntaxe et indentations

- En Python, on n'est pas obligés de finir une ligne par ';'.
  - Mais on peut.
- **Les indentations font partie du langage !**
  - Immense majorité des langages de programmation -> les indentations ne sont là que pour aider les développeurs à lire.
  - En Python, elles font partie de la sémantique du langage.
  - Remplacent les paires d'accolades de beaucoup d'autres langages (C, Java, PHP, JS …).
- Bonne pratique: on indente avec des espaces (4).
  - Se règle dans l'éditeur de texte.

Variables et assignements

- On définit les variables:
  - Sans les typer.
  - En les assignant (pas de déclaration de variable).

```
1 num_items = 4
2 first_name = "John"
3 x = 3.17
```

Variables et assignements

- On définit les variables:
  - Sans les typer.
  - En les assignant (pas de déclaration de variable).

```
1 num_items = 4
2 first_name = "John"
3 x = 3.17
```

- On choisit des noms de variables:
  - explicites et compréhensibles
  - dont la casse/préfixe/suffixe représente le type
  - On évite les double _

Priorités des opérateurs

- Comme dans tout langage de programmation, les opérateurs ont des priorités
- Connaître les priorités de base
- Dans le doute, vérifier ou parenthéser

```
1 x = 3 + 7 if 2 ** 8 == 3 < 2 else  2 + 8 == 7 > 4
2 # que vaut x ?
```

Les chaînes de caractères

- La gestion des chaînes de caractère (strings) en Python est très intuitive

```
1 # On peut créer une string avec des simples '
2 first_name = 'John'
3 # Ou alors avec des double "
4 last_name = "Doe"
5 # Pour créer des longues chaînes de caractères
6 # avec des sauts de ligne, on utilise des triples doubles "
7 python_zen = """
8 Beautiful is better than ugly.
9 Explicit is better than implicit.
10 Simple is better than complex
11 ...
12 """
```

Les chaînes de caractères

- La gestion des chaînes de caractère (strings) en Python est très intuitive

```
1 # On peut créer une string avec des simples '
2 first_name = 'John'
3 # Ou alors avec des double "
4 last_name = "Doe"
5 # On peut utiliser l'opérateur + pour concaténer des strings
6 full_name = first_name + " " + last_name
7 print(full_name)  # -> John Doe
8 print(type(full_name)) # -> <class 'str'>
```

Les chaînes de caractères

- Nombreuses méthodes disponibles

```python
# On peut créer une string avec des simples '
first_name = 'John'
# Ou alors avec des double "
last_name = "Doe"
# On peut utiliser l'opérateur + pour concaténer des strings
full_name = first_name + " " + last_name
print(full_name)  # -> John Doe
print(type(full_name)) # -> <class 'str'>

full_name.startswith(first_name)  # -> True
full_name.endswith("oe")          # -> True
full_name.isascii()               # -> True
full_name.upper()                 # -> 'JOHN DOE'
# ....
```

Les commentaires

On a deux types de commentaires:

- Les commentaires *inline*, dans le code:
  - se démarrent avec un '#'
  - Durent jusqu'à la fin de la ligne
- Les blocs de commentaires:
  - se construisent avec des triples guillemets (""" """ ou ''' ''')
  - utilisés (entre autres) pour documenter les fonctions (*docstring*)
  - disposent de nombreuses fonctionnalités (*doctest)*

La gestion d'erreurs et le debug

- La gestion d'erreur:
  - On peut attraper (*catch*) des exceptions en Python.
  - Par exemple, dans ce bout de code, on itère trop loin dans une liste
  - On déclenche une *IndexError*

```
1 my_list = [1, 2, 3, 4, 5]
2 for i in range(10):
3     print(my_list[i])
4
5 # 1
6 # 2
7 # 3
8 # 4
9 # 5
10 # Traceback (most recent call last):
11 #   File "<stdin>", line 1, in <module>
12 #   File "/home/max/Desktop/toto2.py", line 3, in <module>
13 #     print(my_list[i])
14 # IndexError: list index out of range
```

La gestion d'erreurs et le debug

- La gestion d'erreur:

```python
my_list = [1, 2, 3, 4, 5]
for i in range(10):
    try:
        print(my_list[i])
    except IndexError:
        print("Careful ! You're out of bounds !")
        break  # to exit the 'for' loop
print("Out of the loop")
# -->
# 1
# 2
# 3
# 4
# 5
# Careful ! You're out of bounds !
# Out of the loop
```

Les conditions

- On a les mots-clés *if*, *else* et *elif* qui permettent de créer des branchements conditionels (pas de *switch* ni de *match*).
- Python dispose d'un type booléen natif, *bool*, ainsi que de deux constantes, *True* et *False* (avec une majuscule!)
- On utilise les opérateurs *and*, *or* pour agréger des conditions (if this and that)
- Attention, l'opérateur d'égalité == est différent de l'opérateur d'identité *is*

```
1  x = [1, 2, 3]
2  y = [1, 2, 3]
3
4  x == y  # -> True
5  x is y  # -> False
```

Les fonctions

- On déclare une fonction avec le mot-clé *def*
- On n'a pas besoin de donner les types des arguments et le type de retour des fonctions lorsqu'on les déclare

Les fonctions

- Les arguments d'une fonction peuvent être passés:
  - Dans l'ordre
  - En utilisant des mots-clés

```
1  from random import randrange
2
3  def throw_dice():
4      return 1 + randrange(6)
5
6  throw_dice()  # --> simule le lancer d'un dé
```

Les fonctions

- Les arguments d'une fonction peuvent être passés:
  - Dans l'ordre
  - En utilisant des mots-clés

```
1  from random import randrange
2
3  def throw_dices(num_throws):
4      result = []
5      for i in range(num_throws):
6          result.append(1 + randrange(6))
7      return result
8
9  throw_dices(4)  # --> simule le lancer de 4 dés
```

Les fonctions

- Les arguments d'une fonction peuvent être passés:
  - Dans l'ordre
  - En utilisant des mots-clés

```
1  from random import randrange
2
3  def throw_dices(num_throws):
4      result = []
5      for i in range(num_throws):
6          result.append(1 + randrange(6))
7      return result
8
9  throw_dices(4)  # --> simule le lancer de 4 dés
```

Les fonctions

- Les arguments d'une fonction peuvent être passés, au choix:
  - Dans l'ordre
  - En utilisant des mots-clés
- On peut donner des valeurs par défaut aux arguments

```
1 from random import randrange
2
3 def throw_dices(num_throws=1):
4     result = []
5     for i in range(num_throws):
6         result.append(1 + randrange(6))
7     return result
8
9 throw_dices()  # --> simule le lancer de 1 dé
10 throw_dices(4)  # --> simule le lancer de 4 dés
```

Itérations

- Certains objets et structures de données en Python sont ***itérables***.
  - Par ex, les listes, ensembles, strings, dictionnaires, tuples … sont itérables.

- Pour itérer sur un itérable, on utilise l'opérateur ***for*** **<>** ***in*** **<>:**

```
1 my_list = [2, 4, 6, 8, 10]
2
3 for element in my_list:
4     print(element)
5 # -->
6 # 2
7 # 4
8 # 6
9 # 8
10 # 10
```

Itérations

- Certains objets et structures de données en Python sont ***itérables***.
  - Par ex, les listes, ensembles, strings, dictionnaires, tuples … sont itérables.
  - On peut parfois connaître la taille d'un itérable avec l'opérateur *len*

- Pour itérer sur un itérable, on utilise la s

- Autre possibilité: l'opérateur ***while <>:***

```
1 my_list = [2, 4, 6, 8, 10]
2
3 i = 0
4 while i < len(my_list):
5     print(my_list[i])
6     i += 1
7 # -->
8 # 2
9 # 4
10 # 6
11 # 8
12 # 10
```

Les 4 "grandes" structures de données en Python

- Quatres structures de données centrales en Python
  - Listes (*list*): séquence (mutable) d'éléments.
  - Ensembles (*set*): ensemble d'éléments.
  - Tuples (*tuple*): séquence (immutable) d'éléments
  - Dictionnaires *(dict)*: stocke des paires 'clé, valeur'

- Il existe d'autres structures de données par défaut (frozenset, …)

Listes

- Une liste (*list*) représente une séquence d'éléments.
  - Les éléments peuvent être de n'importe quel type (et de types différents).

- Pour créer une nouvelle liste vide: crochets vides *[]* ou *list()*

```
1  # ces deux variables sont des listes vides
2  empty_list_1 = []
3  empty_list_2 = list()
```

Listes

- Opérateurs et fonctions sur les listes:
  - Concaténation: avec l'opérateur *+.*
  - Ajouter un élément à la fin avec *append.*
- On accède aux éléments via leur indice:
  - Indice positif: depuis la gauche
  - Indice négatif: depuis la droite

```
1 my_list_1 = [2, 4, 6, 8, 10]
2 my_list_2 = [3, 6, 9, 12, 15]
3
4 my_list_1 + my_list_2
5 # --> [2, 4, 6, 8, 10, 3, 6, 9, 12, 15]
6
7 my_list_1.append(12)
8 my_list_1
9 # --> [2, 4, 6, 8, 10, 12]
10
11 my_list_1[0]   # -> 2
12 my_list_1[2]   # -> 6
13 my_list_1[-1]  # -> 12
```

Listes

- Opérateurs et fonctions sur les listes:
  - Concaténation: avec l'opérateur *+.*
  - Ajouter un élément à la fin avec *append.*
- On accède aux éléments via leur indice:
  - Indice positif: depuis la gauche
  - Indice négatif: depuis la droite
- Nombreuses autres fonctions

```
1 my_list_1 = [2, 4, 6, 8, 10]
2 my_list_2 = [3, 6, 9, 12, 15]
3
4 my_list_1 + my_list_2
5 # --> [2, 4, 6, 8, 10, 3, 6, 9, 12, 15]
6
7 my_list_1.append(12)
8 my_list_1
9 # --> [2, 4, 6, 8, 10, 12]
10
11 my_list_1[0]   # -> 2
12 my_list_1[2]   # -> 6
13 my_list_1[-1]  # -> 12
```

Dictionnaires

- Les dictionnaires sont des sortes de listes "généralisées": l'index n'a pas besoin d'être un entier.

- Un dictionnaire associe des clés (*keys*) à des valeurs (*values*).
  - Une seule valeur par clé
  - Les clés doivent être *hashable*

- Les dictionnaires sont un outil extrêmement pratique et utile

Dictionnaires

- On initialise un dictionnaire vide avec *dict()*
  - On peut aussi utiliser *{}*, mais il vaut mieux éviter (confusion avec l'ensemble vide)

```
1  map_name_age = dict()
2
3  map_name_age["John Doe"] = 25
4  map_name_age["Alice Johnson"] = 23
5
6  print(map_name_age)
7  # -> {'John Doe': 25, 'Alice Johnson': 23}
```

- Les clés sont *"John doe"* et *"Alice Johnson"*
- Les valeurs sont 25 et 23

Dictionnaires

Itérer sur un dictionnaire:

- Itérer sur les clés: ***for key in my_dict.keys():***
  - (ou bien *for key in my_dict*)

- Itérer sur les valeurs: ***for value in my_dict.values():***

- Itérer sur les couples clé, valeur: ***for key, value in my_dict.items():***

- Attention! L'ordre n'est pas garanti dans les dictionnaires

Tuples

- Les tuples représentent des séquences d'éléments *non mutables*
  - On ne peut pas ajouter ou supprimer un élément d'un tuple
  - On ne peut pas réassigner un élément dans un tuple

- Pour initialiser un tuple, on sépare les éléments par des virgules
  - On peut (optionnel) entourer les éléments de parenthèses

```
1  my_tuple_1 = 1, 2, 3
2  my_tuple_2 = (4, 5, 6, 7)
3
4  print(my_tuple_1)
5  # -> (1, 2, 3)
6  print(my_tuple_2)
7  # -> (4, 5, 6, 7)
```

Tuples

- On se sert des tuples quand on a une séquence d'éléments dont on connait la taille (fixe), et qu'on ne veut pas modifier.
  - Par exemple: liste d'arguments d'une fonction, couple ou triplet de valeurs ….

- On peut aussi s'en servir pour effectuer des manipulations simples et élégantes

```
1  # on peut échanger (swap) les valeurs
2  # de a et de b en une seule ligne
3  a, b = b, a
4
```

- Les listes ne sont pas *hashables*, mais les tuples le sont

Ensembles

- Un ensemble (*set*) représente un ensemble d'éléments
  - Les éléments doivent être distincts les uns des autres
- On initialise un ensemble vide avec *set().*
  - Attention, *{}* ne représente pas un ensemble vide !

```
1  my_set = set()
2
3  print(my_set)
```

- On peut aussi initialiser un ensemble à partir d'une liste (ou de n'importe quel *iterable*.

Ensembles

- Un ensemble (*set*) représente un ensemble d'éléments
  - Les éléments doivent être distincts les uns des autres
- On initialise un ensemble vide avec *set().*
  - Attention, *{}* ne représente pas un ensemble vide !
- On peut aussi initialiser un ensemble à partir d'une liste (ou de n'importe quel *iterable*.

```
1  my_list = [1, 2, 1, 5, 4, 1, 3, 4]
2  print(my_list)
3  # -> [1, 2, 1, 5, 4, 1, 3, 4]
4  my_set = set(my_list)
5  print(my_set)
6  # -> {1, 2, 3, 4, 5}
```

Ensembles

- Attention ! L'ordre n'est pas garanti dans les ensembles !

- On se sert des ensembles quand:
  - On veut être sûr de ne pas avoir de doublons.
  - On veut pouvoir trouver, ajouter ou supprimer rapidement des éléments.

- Nombreuses opérations disponibles sur les ensembles:
  - Union (|), intersection (&), relation d'inclusion (<) ….

- Les éléments d'un ensemble doivent être *hashable*

Les "comprehensions"

Les *comprehensions* sont une syntaxe particulière à Python

- Permettent d'initialiser élégamment des listes, ensembles, tuples ou dictionnaires
- On parle alors de *list comprehension, set comprehension, dict comprehension, tuple comprehension*

List comprehensions

La syntaxe d'une list comprehension est la suivante:

- **[f(x) for x in some_iterable]**

```
# On peut créer la liste des premiers
# multiples de 7 comme ça:
my_list = []
for n in range(12):
    my_list.append(7*n)

print(my_list)
# -> [0, 7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77]

# On peut utiliser une liste comprehension:
# plus court, plus lisible
my_other_list = [7*n for n in range(12)]
print(my_other_list)
# -> [0, 7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77]
```

Autres comprehensions

Le principe est le même pour les autres structures de données:

- Avec des parenthèses pour les tuples
- Avec des accolades pour les sets
- Avec la syntaxe {***key: value for x in some_itable}***

- Avantages des comprehensions:
  - Plus court
  - Plus rapide à exécuter
  - Plus lisible

Autres structures de données

On trouve d'autres structures de données utiles en Python:

- Les *frozenset* sont une version immutable et hashable des sets
- Le module *collections* contient de nombreuses structures utiles:
  - ***collections.deque***: implémentation d'une pile (*stack*)
  - ***collections.defaultdict:*** implémentation d'un dictionnaire donnant une valeur par défault aux clés n'existant pas
  - ***collections.OrderedDict***: dictionnaire conservant l'ordre
  - ***collections.namedtuple***: tuple dont les éléments possèdent un identifiant
  - Etc ….

Le "duck typing"

- Le typage en Python est particulier: on parle de *duck typing*
  - Jeu de mot -> duck tape (gaffeur).
  - *"If it walks like a duck and talks like a duck, it must be a duck".*

- Le duck typing est une forme "souple" de typage:
  - On vérifie durant l'éxécution que les objets ont les bonnes propriétés, sans vérifier leur type.
  - On verra cette notion plus en détail au fil du semestre.

- À la fois un point fort de Python (plus facile de programmer) et un point faible (plus facile de faire des erreurs).

Gardez en tête que:

- Cette présentation est très superficielle: on n'a fait que survoler quelques incontournables de Python.

- On va, au cours du semestre, voir des concepts qui permettent de mieux comprendre comment ce langage fonctionne.

- La documentation Python est très bien faite

Pour la semaine prochaine

- Téléchargez le notebook d'exercice de la 1ère séance.

- Essayez de faire un maximum d'exercices pour vous faire la main.

- Au programme la semaine prochaine:
  - Les fondements de la programmation orientée objet.
  - Les objets, classes, instances en Python.
  - Attributs et méthodes.