EPFL

ICC

Programmation

GC/MX, **Cours 5**

21 octobre 2022

Jean-Philippe Pellet

Opening music: Mark Knopfler, album *Tracker* (2015)

Previously, on Programmation...

- Types de base en Python: int, float, str, bool
- Méthodes, fonctions et slicing pour calculer des valeurs dérivées
- Conditions pour exécuter du code selon la valeur d'une expression booléenne:
  `if <condition>: ... else: ...` et ses variantes
- Boucles pour exécuter du code plusieurs fois:
  - Boucle `while <condition>: ...`
  - Boucle `for i in range(...): ...`
- Déclaration de fonctions avec type de retour et paramètres:
  - `def calculate_area(r: float) -> float: return ...`
- Utilisation de listes:
  - `values: List[int] = [1, 4, 2, 7, 3]`

Slicing

```
my_ints = [10, 20, 30, 40, 50, 60]
my_ints[0]           # 10

my_ints[0:2]         # [10, 20]

my_ints[:4]          # [10, 20, 30, 40]

my_ints[4:]          # [50, 60]

my_ints[4:] = [55, 65]
my_ints              # [10, 20, 30, 40, 55, 65]

my_ints[4:] = []
my_ints              # [10, 20, 30, 40]

my_ints[0:0] = [0, 1, 2]
my_ints              # [0, 1, 2, 10, 20, 30, 40]

my_ints[0] = [0, 1, 2]
my_ints              # [[0, 1, 2], 1, 2, 10, 20, 30, 40]
```

une seule position — déjà connu

une sous-liste de 0 (inclus) à 2 (exclu)

une sous-liste jusqu'à 4 (exclu)

une sous-liste depuis 4 (inclus)

remplacement d'une sous-liste

remplacement d'une sous-liste par liste vide ⇒ suppression d'éléments contigus

remplacement d'une sous-liste vide par une liste non-vide ⇒ insertion d'éléments contigus

[0:0] n'est pas la même chose que [0], qui désigne une case précise

Répétition — Listes

Déclarez une liste avec les int de 0 (incl.) à 5 (excl.)

```
from typing import List
values: List[int] = [0, 1, 2, 3, 4]
values: List[int] = list(range(5)) # idem!
```

Ajoutez 1 à la valeur de la première case

```
values[0] += 1          # [1, 1, 2, 3, 4]
```

Supprimez les deux premières cases

```
values[0:2] = []        # [2, 3, 4]
```

Ajoutez les deux valeurs –2 et –3 entre 2 et 3

```
values[1:1] = [-2, -3] # [2, -2, -3, 3, 4]
```

Ajoutez 42 à la fin de la liste

```
values.append(42)       # [2, -2, -3, 3, 4, 42]
```

Effacez toute la liste

```
values.clear()          # []
```

Test case: Problème des 100 prisonniers

- 100 prisonniers dans des cellules numérotées de 0 à 99
- On prend les clés qu'on cache dans des boîtes numérotées de 0 à 99
- Chaque prisonnier peut ouvrir 50 boîtes (sans savoir ce qu'on fait les autres prisonniers)
- Si tous les prisonniers trouvent leur clé, ils peuvent tous partir.
  - Quelle meilleur stratégie, pour quelle probabilité de sortie?
    - ➡ Approche **naïve**: $1/2^{100}$
    - ➡ Approche **optimale**: environ 31%
  - https://www.youtube.com/watch?v=iSNsgj1OCLA

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Vous n'y croyez pas? Simulons ce problème!

Test case: Problème des 100 prisonniers

```python
from random import shuffle
n_prisoners = 100
n_attempts = 50

def simulate_one() -> bool:
    boxes = list(range(n_prisoners))
    shuffle(boxes)

    for p in range(n_prisoners):
        found_number = boxes[p]
        can_still_open = n_attempts - 1

        while can_still_open > 0 and found_number != p:
            found_number = boxes[found_number]
            can_still_open -= 1

        if found_number != p:
            return False

    return True
```

```python
n = 1000
successes = 0
for i in range(n):
    if simulate_one():
        successes += 1

print(successes / n)
```

Lance une simulation, et retourne *True* si les prisonniers ont réussi à sortir

Crée une liste de 0 à 99 et la mélange aléatoirement (reproduit le fait de cacher les clés)

Répète ceci pour chaque prisonnier avec *p* = numéro de cellule du prisonnier

On va voir le numéro derrière la boîte *p* et on se rappelle qu'on a le droit d'ouvrir encore 49 autres boîtes

Tant qu'on n'a pas trouvé la bonne clé et qu'on a encore le droit d'ouvrir d'autres boîtes, on le fait

Si le prisonnier n'a pas trouvé sa clé, c'est fichu pour tout le monde: on peut faire un *return False* directement depuis la boucle

Si on arrive ici, c'est que tous les prisonniers ont trouvé leur clé!

Pour avoir une estimation de probabilité, on répète l'expérience *n* fois et on compte le nombre de succès

Évaluation du cours

- Donnez une note au cours sur IS-Academia
  - Dites ce qui vous a plu, ce qui vous a déplu; ce qui vous convient, ce qui ne vous convient pas
- Feedback anonyme
  - Mais soyez constructifs: tout le monde en profitera
- Discussion des résultats dès que nous les obtenons
- Vos commentaires — indiquez svp. pour quelle partie vous faites un commentaire:
  - «**théorie**» → partie d'Olivier Lévêque
  - «**python**» → partie programmation

Cours de cette semaine

Objets immuables ou modifiables

Imports

Sets

Motivation: question

```
def modify(ws: List[str]) -> None:
    ws[1] = "beau"
    ws[2] = "il"
```

```
words = ["fort", "belle", "elle", "dort"]
print(words)
modify(words)
print(words)
```

Paul de Rességuier,
Épitaphe d'une jeune fille

```
words = ...
print(words)
ws = words
ws[1] = "beau"
ws[2] = "il"
print(words)
```

Qu'affiche le second print()?

```
['fort', 'beau', 'il', 'dort']
```

Motivation: question

```
def modify(v: int) -> None:
    v = v + 4

value = 42
print(value)
modify(value)
print(value)
```

```
value = 42
print(value)
v = value
v = v + 4
print(value)
```

Qu'affiche le second print()?

42

But why??

Objet immuable, objet variable

- En Python, on peut classifier les valeurs qu'on donne aux variables en deux

  — Les objets immuables ne changent pas intrinsèquement

  ➡ `int`, `float`, `str`, `bool`

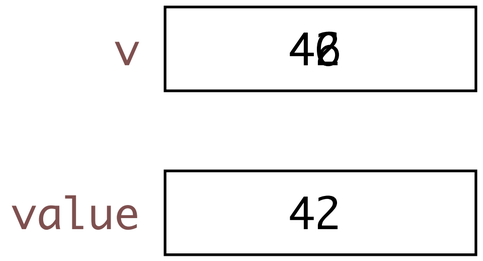
  — Les objets variables peuvent changer via des appels de méthodes, slicing, opérateurs, etc.

  ➡ `List`, `Set`, `Dict`

Avec un objet immuable

```
def modify(v: int) -> None:
    v = v + 4

value = 42
print(value)  # 42
modify(value)
print(value)  # toujours 42
```

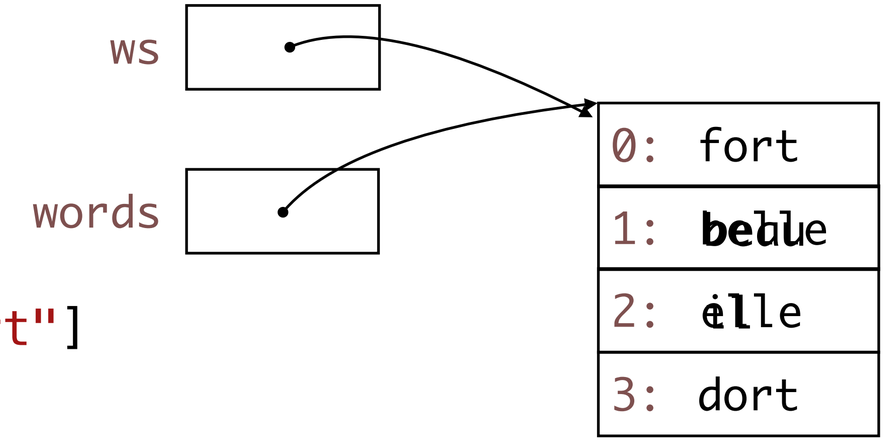


int est un type dont les objets sont immuables. On ne les modifie pas directement, mais on crée de «nouveaux ints» à chaque opération. Réaffecter une variable locale comme `v` *ne change pas* `value`.

Avec un objet modifiable

```
def modify(ws: List[str]) -> None:
    ws[1] = "beau"
    ws[2] = "il"

words = ["fort", "belle", "elle", "dort"]
print(words)
modify(words)
print(words) # ['fort', 'beau', 'il', 'dort']
```



List est un type dont les objets sont modifiable. On peut manipuler directement leur contenu. Les modifications sont vues par toutes les références au même objet.

Modifier l'immuable?

Comment modifier une variable d'un type immuable?

On peut le faire seulement indirectement en Python.

1. Il faut retourner la valeur modifiée depuis la méthode

```
def modify2(value: int) -> int:
    return value + 4
```

2. Il faut réassigner la valeur de retour à la variable à modifier lors de l'appel

```
value = 42
print(value)   # 42
value = modify2(value)
print(value)   # 46
```

Empêcher de modifier le modifiable?

Comment être sûr qu'un appel de fonction ne va pas modifier une structure modifiable comme une liste?

On peut passer une copie de la structure..

```
def modify(ws: List[str]) -> None:
    ws[1] = "beau"
    ws[2] = "il"
```

Même fonction qu'avant

```
words = ["fort", "belle", "elle", "dort"]
print(words)
modify(words.copy())
print(words)
            # ['fort', 'belle', 'elle', 'dort']
```

C'est une copie de la liste qui est fournie. L'original reste intact

Cours de cette semaine

Objets immuables ou modifiables

Imports

Sets

Import

```
import math
print(math.cos(math.pi))
```

Tout ce qui est défini dans math.py est accessible avec «math.xxx»

```
import math as m
print(m.cos(m.pi))
```

Tout ce qui est défini dans math.py est accessible avec «**m**.xxx»

```
from math import *
print(cos(pi))
```

Tout ce qui est défini dans math.py est accessible directement

```
from math import pi, cos as cosine
print(cosine(pi))
```

Seulement *pi* et *cos*, définis dans math.py, sont importés; *cos* est renommé *cosinus*

```
from typing import List
```

Aussi valable pour les types

Partager du code entre plusieurs fichiers

- Chaque fichier *.py* est un module
- Vous pouvez donc créer et importer vos propres modules

Dans *mytools.py*

```
from typing import List

def double(values: List[int]) -> List[int]:
    return [2 * x for x in values]

def make_string(values: List[int], separator: str = ", ") -> str:
    return separator.join([str(x) for x in values])
```

Dans un autre fichier du même dossier

```
from mytools import *
print(double([1, 2, 3]))
print(make_string([1, 2, 3], separator=" -> "))
```

Variante:

```
from mytools import double as dbl
print(dbl([1, 2, 3]))
```

on renomme une fonction

Cours de cette semaine

Objets immuables ou modifiables
Imports
Sets

Set

- Un set (ensemble) est similaire à une liste, mais
- … n'a pas d'ordre intrinsèque
  - Pas possible d'utiliser l'indexation [i] ou le slicing [x:y]
- … contient un élément au plus une fois
  - et permet de tester rapidement s'il contient un élément ou non
- Objets modifiables (non immuables) comme les listes

Comparaison List/Set

```python
from typing import List, Set


my_list: List[str] = ["bonjour", "hello", "bonjour"]
print(len(my_list))    # 3
print(my_list[2])      # 'bonjour'
my_list = []           # liste vide


my_set: Set[str] = {"bonjour", "hello", "bonjour"}
print(len(my_set))     # 2
#print(my_set[2])      # pas possible, le set n'a pas d'ordre
my_set = set()         # set vide - pas {}


my_set = set(my_list)  # conversion de liste en set
my_list = list(my_set) # conversion de set en liste
```

Listes: avec la notation [ ]

Sets: avec la notation { }

Éléments dupliqués ne sont pas ajoutés une seconde fois

Autres méthodes utiles sur les sets

- `my_set.add(x)` — ajouter un élément *(listes: `append(x)`)*
- `my_set.clear()` — tout effacer
- `my_set.remove(x)` — supprime `x`
- `x in my_set` — teste si my_set contient
  - `if x in my_set: ...`
  - `if x not in my_set: ...`
- Méthodes pour l'union, l'intersection ou encore la différence de plusieurs sets
  - Serait aussi possible avec les listes, mais plus lent qu'avec les sets
    - ➡ Représentation interne différente

Résumé Cours 5

- Les objets immuables ne peuvent pas être modifiés; les objets modifiables peuvent subir des modifications (*Yay…*)
- On utilise import ou from… import pour réutiliser du code défini dans un autre fichier
- Un set (de type `Set[T]`) est un peu comme une liste, mais assure l'unicité des éléments
  - Pas d'ordre intrinsèque; on ne peut pas récupérer l'élément *i*
  - On peut convertir entre des listes et des sets facilement