Types construits

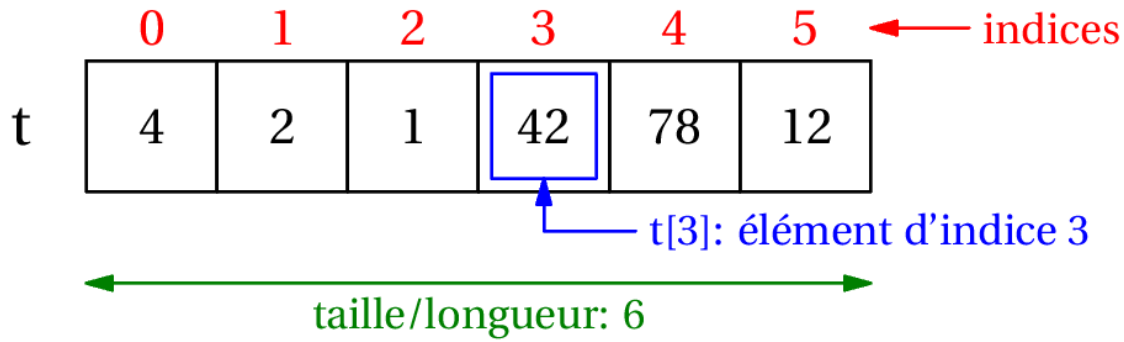
Les tuples et les dictionnaires

Les structures de données abordées

1. Les tableaux
2. Les p-uplets
3. Les dictionnaires / p-uplets nommés

Les tableaux (listes)

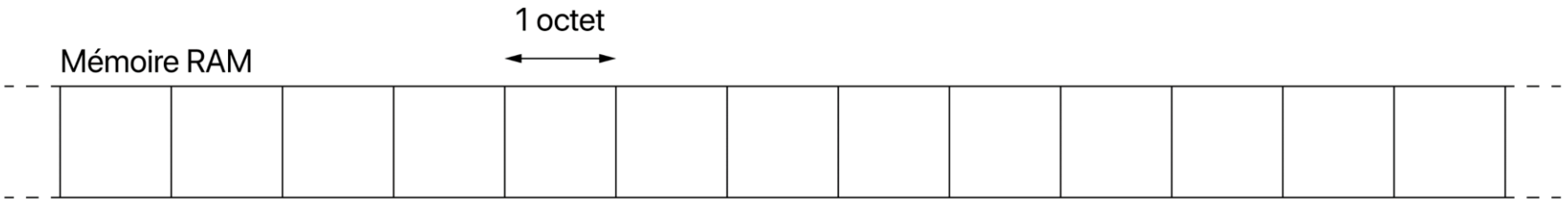
- Un tableau est un suite **continue et ordonnée** d'éléments de **même type.**



- **Continue :** les éléments se suivent dans la mémoire vive.
- **De même type :** les éléments sont de même taille et sont interprétés de la même manière.
- **Un tableau** est de taille **variable** (on peut ajouter ou enlever des éléments).

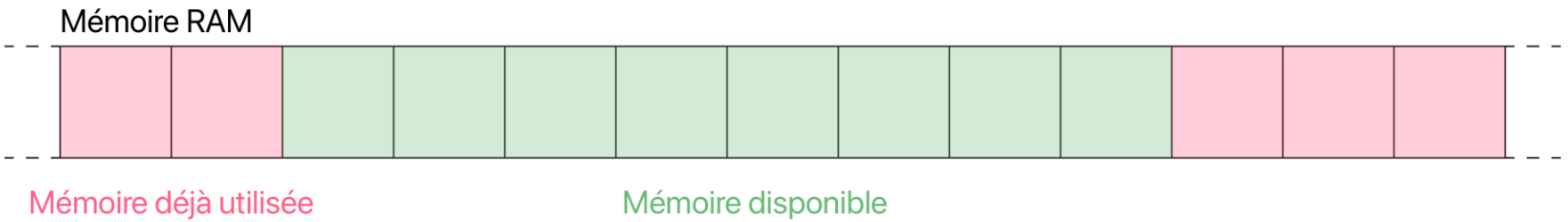
L’allocation de mémoire

La mémoire RAM de votre ordinateur n’est qu’un très grand tableau découpé en octet :



L’allocation de mémoire

Certaines parties de cette mémoire sont déjà utilisées par d’autres programmes :



L’allocation de mémoire

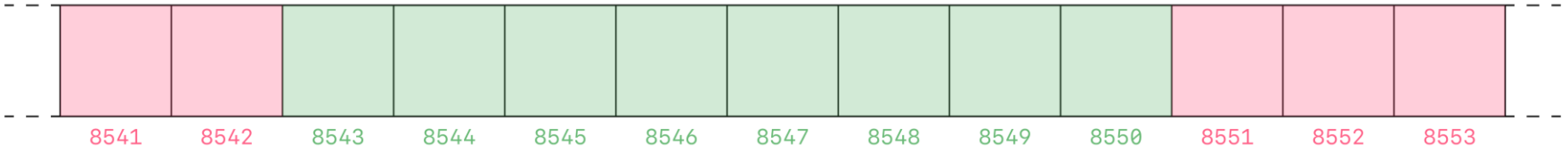
Imaginons, je souhaite stocker 3 entiers de 16-bits dans la mémoire :

```
tab = [42, 13, 128]
```

Que se passe-t-il ?

1. Python demande à l’OS « Je voudrais 6 octets »

Mémoire RAM

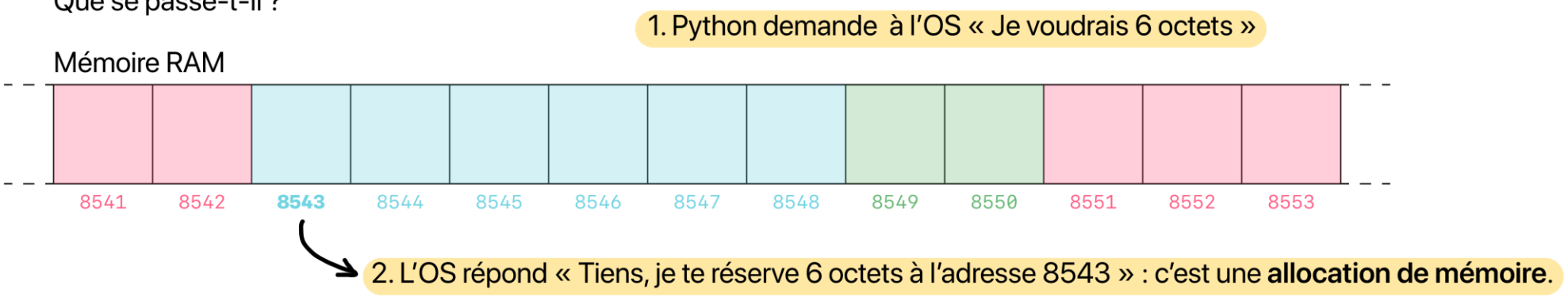


L’allocation de mémoire

Imaginons, je souhaite stocker 3 entiers de 16-bits dans la mémoire :

```
tab = [42, 13, 128]
```

Que se passe-t-il ?



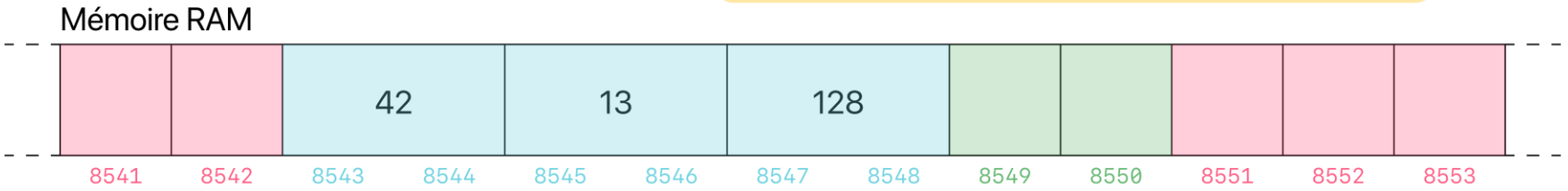
L’allocation de mémoire

Imaginons, je souhaite stocker 3 entiers de 16-bits dans la mémoire :

```
tab = [42, 13, 128]
```

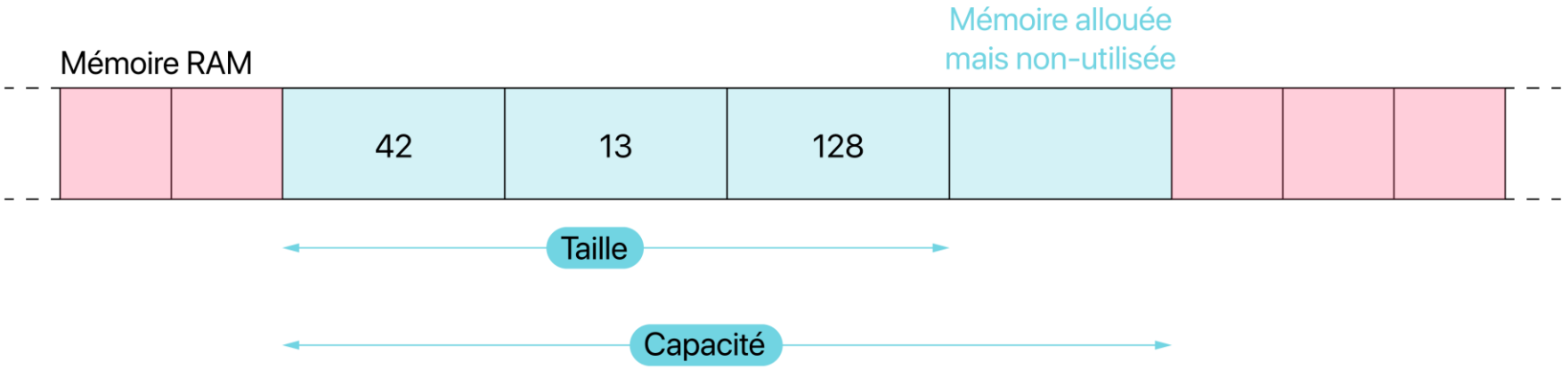
Que se passe-t-il ?

3. Python copie les valeurs à l’emplacement indiqué

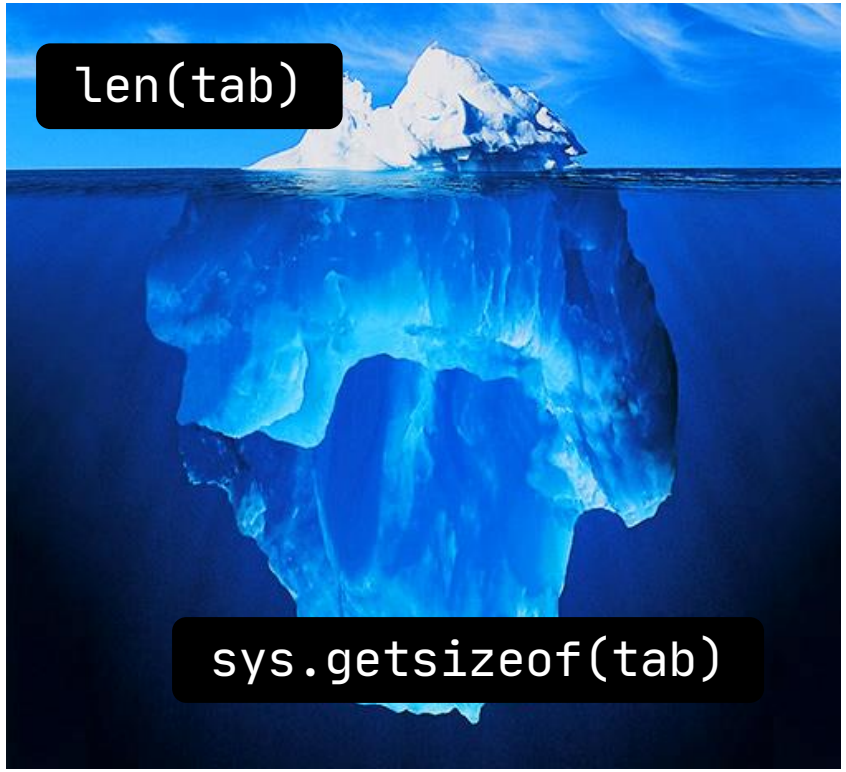


L’allocation de mémoire

En pratique, Python demande une plus grande allocation de mémoire.
Ce surplus de mémoire permet d’éviter les réallocations lorsqu’on ajoute un élément.



La capacité d'un tableau



- En pratique, un tableau possède une **taille** et une **capacité,** c'est-à-dire un peu de mémoire supplémentaire au cas où on lui ajoute des éléments.
- En effet, si l'on dépasse la capacité en ajoutant un élément, alors on effectue une **réallocation** dans la mémoire (le programme réserve un emplacement plus grand dans la mémoire et copie le tableau).
- En pratique, la capacité est environ 1.5 fois la taille de la liste.

Les p-uplets (tuples)

- Un n-uplet, ou tuple, est une séquence **continue et ordonnée** d'éléments de **types possiblement différents.**

```
eleve = ("Michel", "Dupont", 17, 175)
print(eleve[0])  # affiche Michel
```

Un tuple se construit avec des parenthèses !

L'accès à un élément se fait encore par indice

- **Immutable :** un tuple n'est généralement pas modifiable (par exemple en Python). Il n'est par contre jamais possible d'ajouter ou de retirer des éléments à un tuple.

```
eleve[0] = "Pierre"
# erreur ('tuple' object does not support item assignment)
```

Les p-uplets (tuples)

- Quelques exemples de tuples :

```
mon_tuple = () # tuple vide

mon_tuple = (1, 2, 3) # tuple avec des entiers

mon_tuple = (1, "Hello", 3.4) # tuple avec des types différents

mon_tuple = ("carabistouille", [8, 4, 6], (1, 2, 3))
```

Les p-uplets (tuples)

- Opérateur de concaténation `+` marche comme pour les listes :

```
t1 = (12, 34, 56)
t2 = (78, 90)
print(t1 + t2) # affiche (12, 34, 56, 78, 90)
```

- Opérateur de répétition `*` marche aussi :

```
t1 = (12, 34, 56)
print(t1 * 3) # affiche (12, 34, 56, 12, 34, 56, 12, 34, 56)
```

- Opérateur de comparaison (`==`, `!=`, `>`, etc.) et d'appartenance (`in`, `not in`)
- Un tuple est un itérable donc on peut aussi le parcourir par éléments/indices.

Attends, mais c'est quoi la différence avec une liste en fait ?

```
import sys
une_liste = [123, 231, 13.31, 0.1312]
un_tuple  = (123, 231, 13.31, 0.1312)
print('Taille de la liste :', sys.getsizeof(une_liste), 'octets')
print('Taille du tuple :   ', sys.getsizeof(un_tuple) , 'octets’)

# Affiche :
# Taille de la liste : 88 octets
# Taille du tuple :    72 octets
```

- Comme on ne peut ni ajouter ni retirer d'éléments à un tuple, il n'y a pas besoin de réserver plus que la taille totale des éléments. Un tuple n'a pas de **capacité**. C'est donc une structure de données plus compacte et donc plus rapide !

« Unpacking a tuple »

```
un_tuple = ('Chat', True, 10.42, 10)

a, b, c, d = un_tuple

print(b)  # affiche True
```

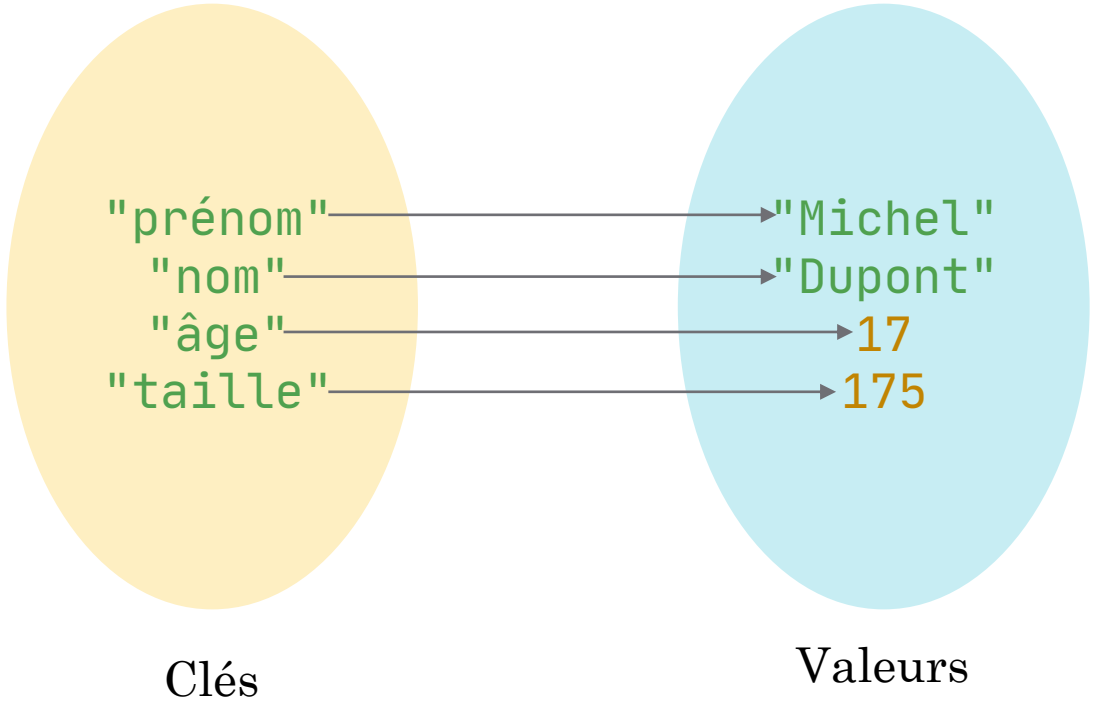
- Ca marche aussi pour les listes… et n'importe quel itérable en fait !
- Quand vous écrirez :

  ```
  a, b, c, d = 10, 20, 30, 40
  ```

  Vous créez implicitement un tuple (10, 20, 30, 40) que vous « déballez » dans les variables a, b, c et d.
- Sauriez-vous échanger le contenu de deux variables avec cette technique ?

Les dictionnaires

- Un dictionnaire une structure de données non-ordonnée qui permet une **association clé–valeur**.



Les dictionnaires

- Un dictionnaire une structure de données non-ordonnée qui permet une **association clé–valeur**.

```
eleve = {
    "prénom": "Michel",
    "nom": "Dupont",
    "âge": 17,
    "taille": 175
}
```

Un dictionnaire se construit avec des accolades !

Les dictionnaires

- Un dictionnaire une structure de données non-ordonnée qui permet une **association clé–valeur**.

```
eleve = {
    "prénom": "Michel",
    "nom": "Dupont",
    "âge": 17,
    "taille": 175
}

print(eleve["prénom"])
print(eleve["nom"])
print(eleve["âge"])
print(eleve["taille"])
```

Exercice

- Ecrire le programme qui affiche le prix d'une orange :

```
produits = {'pomme': 0.40, 'orange': 0.35, 'banane': 0.25}
```

Exercice

- Ecrire le programme qui affiche le prix d'une orange :

```
produits = {'pomme': 0.40, 'orange': 0.35, 'banane': 0.25}
print(produits['pomme'])
```

Les opérations de base

- Trois opérations de base :
  - Créer un dictionnaire vide

```
dico = {}
```

  - Ajouter/modifier une association

```
dico['ma_clé'] = 40
dico['ma_clé'] = 52
```

  - Lire la valeur associée à une clé

```
print(dico['ma_clé']) # affiche ?
print(dico['bonjour']) # erreur
```

Exercice

- Suite à l'inflation, le prix d'une pomme a doublé ce mois-ci, modifiez sa valeur ! Et ajoutez une pêche à 1,40 € !

```
produits = {
    'pomme': 0.40,
    'orange': 0.35,
    'banane': 0.25,
}
```

Opérations utiles

- D'autres opérations :
  - Savoir si une clé existe dans le dictionnaire :

```
cle in dico
```

  - Supprimer une association :

```
del dico[cle]
valeur = dico.pop(cle)
```

  - Retourne un liste de l'ensemble des clés :

```
dico.keys()
```

Exercice

- Il n'y a plus de pétrole pour importer les bananes supprimez-moi ça !

```
produits = {
    'pomme': 0.40,
    'orange': 0.35,
    'banane': 0.25,
    'pêche': 0.55,
    'nectarine': 0.7,
    ... # d'autres associations
}
```

- Euh, attend, on a bien mis les poires ? Tu peux vérifier ?

Itérer sur un dictionnaire

```python
# parcours par clé (très courant)
for key in dico:
    print(key, dico[key])

# parcours par clé (moins joli)
for key in dico.keys():
    print(key, dico[key])

# parcours par valeurs (plutôt rare)
for value in dico.values():
    print(value)

# parcours par clé et valeurs (j'aime bien)
for key, value in dico.items():
    print(key, value)
```

Exercice

- En fait, c'est plus grave que ça ! L'inflation fait gonfler le prix de tous les articles, multiplie tous les prix par deux !

```
produits = {
    'pomme': 0.40,
    'orange': 0.35,
    'banane': 0.25,
    'pêche': 0.55,
    'nectarine': 0.7,
    ... # d'autres associations
}
```

- Quel est le fruit le plus cher ?

Exercice

```
produits = {
    'pomme': 0.40,
    'orange': 0.35,
    'banane': 0.25,
    'pêche': 0.55,
    'nectarine': 0.7,
    ... # d'autres associations
}
```

- Combien de fruits sont en dessous de 60 centimes ?

Les dictionnaires

- Les clés doivent être impérativement du **même type**.
- Le **fonctionnement** interne d'un dictionnaire n'est pas au programme (c'est complexe).
- Finalement, cette structure de données peut se voir comme une **extension** des tableaux (un tableau utilise des nombre entiers positifs et continues comme clés, les indices).
- Très utile pour traiter les **données en table** !
- Notion de « **n-uplet nommé** » : à mi-chemin entre un tuple et un dictionnaire. Il permet de nommer explicitement chacun des éléments du tuple, au lieu d'utiliser des indices. L'avantage d'un n-uplet nommé sur un dictionnaire est que l'on garde la **continuité** et la linéarité dans la mémoire (en Python, il faut normalement importer la bibliothèque « `namedtuple` » pour définir cette structure de données).