Structures de données

- types simples
  - nombres entiers : types **`int`** et **`long`**
  - nombres non entiers : type **`float`**
  - valeurs booléennes (ou logiques) : type **`bool`** (deux valeurs : **`False`**, **`True`**)
  - « absence de valeur » : type **`NoneType`** (une seule valeur : **`None`**)
- types complexes offerts par le langage et la bibliothèque
  - *list* **`[`** α**`,`** β**`,`** χ **`]`**
    accès indexé, ajout et suppression d'éléments permis
  - *tuple* **`(`** α**`,`** β**`,`** χ **`)`**
    (n-uplet) accès indexé, encombrement optimisé
    Immuable : ajout, suppression et remplacement interdits
  - *string* **`"`***abc*`"`**, **`'`***abc*`'` ou **`"""`***abc*`"""`**
    suite *immuable* de caractères
  - *set* **`set([`** α**`,`** β**`,`** χ **`])`**
    non-répétition des éléments (pas d'indexation)
    accès par valeur (test d'appartenance) optimisé
  - *map***`{`** α **`:`** A**`,`** b **`:`** B**`,`** χ **`:`** X **`}`**
    (table associative) : collection de couples (*clé*, *valeur*)
    accès par clé optimisé

Structures de données

- types simples
  - nombres entiers : types **`int`** et **`long`**
  - nombres non entiers : type **`float`**
  - valeurs booléennes (ou logiques) : type **`bool`** (deux valeurs : **`False`**, **`True`**)
  - « absence de valeur » : type **`NoneType`** (une seule valeur : **`None`**)
- types complexes offerts par le langage et la bibliothèque
  - *list* accès indexé, ajout et suppression d'éléments permis
  - *tuple* (n-uplet) accès indexé, encombrement optimisé
    Immuable : ajout, suppression et remplacement interdits
  - *string* suite *immuable* de caractères
  - *set* non-répétition des éléments (pas d'indexation)
    accès par valeur (test d'appartenance) optimisé
  - *map*(table associative) : collection de couples (*clé*, *valeur*)
    accès par clé optimisé
- structures à monter soi-même
  - *file* (*queue*) structure évolutive *FIFO* (first in first out)
  - *pile* structure évolutive *LIFO* (last in first out)
  - *arbre* organisation hiérarchique
  - *graphe* représentation de relations entre éléments

Gestion de la mémoire

- bonne nouvelle :
  - les structures naissent et grossissent automatiquement
  - il n'y a rien pour leur destruction
  - Python gère la [pénurie de] mémoire : le programmeur l'utilise comme si elle était infinie

```
liste = [ "Anne", "Bernard", "Carole", "Denis" ]
...
```
utilisation de la liste des personnes
```
...
liste = [ 11, 22, 33, 44, 55, 66, 77, 88 ]
...
```
ici, la première liste est construite mais n'est plus référencée

- mécanisme du *garbage collector* :

  activation automatique lorsque la mémoire manque :

  1° parcours des structures auxquelles les *variables accessibles* réfèrent en marquant les cellules de mémoire que ces structures occupent

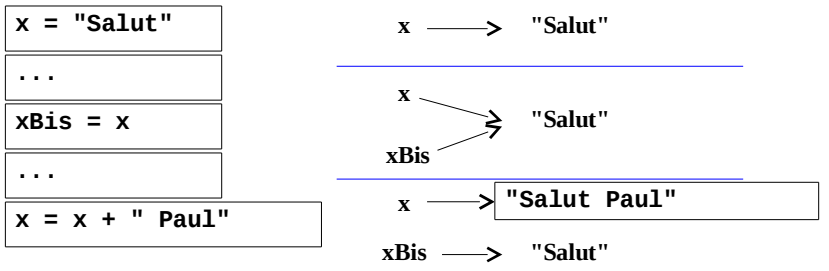
  2° parcours de toute la mémoire
  en inscrivant les cellules non marquées comme réutilisables

Listes : désignées par référence et modifiables

- le schéma suivant est-il fiable ?

```
x = expression
...
xBis = x                    # duplication de la valeur actuelle de x
opérations modifiant x
```
ici **`xBis`** *a la valeur initiale de* **`x`**

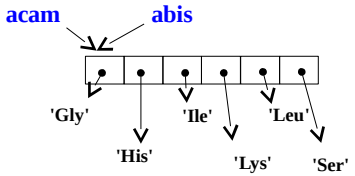
- c'est vrai si la valeur de *x* est immuable (nombres, chaînes…)



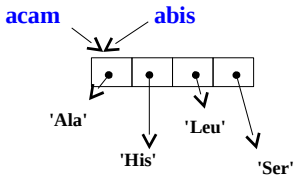
Listes : désignées par référence et modifiables

- exemple :

```
>>> aa = [ 'Gly', 'His', 'Ile', 'Lys', 'Leu', 'Ser']
>>> aaBis = aa          ## aaBis sauvegarde de aa ?
>>> del aa[2:4]
>>> aa[0] = "Ala"
>>> aa
['Ala', 'His', 'Leu', 'Ser']
>>> aaBis
['Ala', 'His', 'Leu', 'Ser']
>>>
```



aaBis *a été modifié comme* **aa**



Opérations sur les listes

- accès (opérations disponibles sur toutes les « séquences »)
  - **len(***liste***)** longueur
  - *élt* **in** *liste* appartenance
  - *liste*$_1$ **+** *liste*$_2$ concaténation
  - *liste***[***i***]** élément de rang *i* (attention au débordement)
  - *liste***[***i***:***j***]** tranche de *i* à *j*-1 (*i* absent : début, *j* absent : fin) *renvoie une liste nouvelle*
  - *liste***[***i***:***j***:***k***]** tranche de *i* à *j*-1 de *k* en *k*
- modification
  - *liste***[***i***] =** *élt* remplacement d'un élément existant (att. au débordement)
  - *liste***[***i***:***j***] =** *liste* remplacement d'une sous-liste par une liste *peut servir à allonger ou raccourcir la liste*
  - **del** *liste***[***i***:***j***]** suppression d'une sous-liste
  - *liste***.append(***élt***)**, *liste***.extend(***liste*$_2$**)** ajout d'élément, concaténation de *liste*$_2$
  - *liste***.count(***élt***)**, *liste***.index(***élt***)** nombre et position de *élt* dans *liste*
  - *liste***.pop(***i***)**, *liste***.pop()** supprime et renvoie l'élément de rang *i* [resp. le dernier]
  - *liste***.reverse()** renverse l'ordre des éléments
  - *liste***.sort()**, *liste***.sort(***compar***)** tri de la liste [resp. avec le critère *compar*]

Listes – Distinguer :

- les éléments et les [sous-]listes

```
liste = [ 11, 22, 33, 44, 55 ]
```

suivi de :

```
liste[2:3] = 0
```
ERREUR (il faut une liste)

```
liste[2:3] = [ 0 ]
```
OK, la liste devient `[ 11, 22, 0, 44, 55 ]`

```
liste[2] = 0
```
OK, la liste devient `[ 11, 22, 0, 44, 55 ]`

```
liste[2] = [ 0 ]
```
OK, la liste devient `[ 11, 22, [ 0 ], 44, 55 ]`

- les opérations *construisant une nouvelle liste* et les opérations *sur place*

```
>>> l = [3, 9, 6, 1, 5, 10, 7, 2, 4, 8]
>>> sorted(l)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> l
[3, 9, 6, 1, 5, 10, 7, 2, 4, 8]

>>> l.sort()
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

construit une nouvelle structure n'altère pas l'original

optimise temps et espace

Tuples

- en « français » *n-uplets* (comme dans *tri*plets, *quadru*plets, etc.)
- comme les tableaux d'autres langages, rangement compact :
  - le nombre d'éléments est figé
  - l'accès indexé est optimisé
- syntaxe
  - trois cas : plus de deux éléments, un élément, aucun élément :

```
>>> t5 = ( 11, 22, 33, 44, 55 )
>>> t5, type(t5)
((11, 22, 33, 44, 55), <type 'tuple'>)

>>> t1 = ( 11, )
>>> t1, type(t1)
((11,), <type 'tuple'>)

>>> t0 = ( )
>>> t0, type(t0)
((), <type 'tuple'>)

>>>
```

ce qui est bleu est facultatif

parenthèses obligatoires

Tuples

- exemple : manipuler plusieurs valeurs comme une seule

```
def minMaxMoy(liste):
    " renvoie le min, le max et la moyenne de la liste "
    n = len(liste)
    if n == 0:
        return None
    som = 0
    min = max = liste[0]
    for x in liste:
        if x < min:
            min = x
        elif x > max:
            max = x
        som = som + x
    return (min, max, som / n)
```

- emploi de cette valeur, 2ème manière

```
min, max, moy = minMaxMoy([ 10, 18, 14, 20, 12, 16 ])

print(min, max, moy)
```


Transformation terme à terme d'une liste

- construction d'une liste par transformation des éléments d'une autre. Mécanisme dit *list comprehension* :

[*expression* **for** *variable* **in** *liste* **]**

- exemple : multiplier chaque élément d'une liste de nombres

```
>>> liste1 = [ 11, 22, 33, 44, 55, 66, 77, 88, 99 ]
>>> liste2 = [ 10 * x  for x in liste1 ]
>>> liste2
[ 110, 220, 330, 440, 550, 660, 770, 880, 990 ]
>>>
```

- attention :
  la liste obtenue a autant d'éléments que l'original, donc
  - cela ne peut servir ni à filtrer une liste,
  - ni à calculer un cumul

List comprehension

- Une autre façon de construire une liste
- Syntaxe compacte
- Très proche d'une expression mathématique

En algèbre : $S = \{ 2x \mid x \in \mathbb{N}, x^2 > 30 \}$ dénote un ensemble infini.

expression de sortie: $2x$

variable libre : x

ensemble d'entrée : $\mathbb{N}$

prédicat : $x^2 > 30$

En python, nous avons la possibilité de construire des ensembles (ordonnés) finis :

```
S = [ 2*x for x in range(0, 1000) if x*x > 30 ]
```


List comprehension

- Équivalence avec une boucle for :

```
S = [ 2*x for x in range(0, 1000) if x*x > 30 ]
```

est équivalent à :

```
S = []
for x in range(0, 1000):
    if x*x > 30:
        S.append(2*x)
```

Set

- collection sans répétition d'éléments immuables
- pas de syntaxe spécifique pour construire : appliquer **`set`** à un *itérable*

```
>>> l = [ 11, 22, 11, 22, 33, 11, 22, 33, 44 ]
>>> s = set(l)
>>> s
set([33, 11, 44, 22])
>>>
```

l'ordre des éléments n'a pas de signification

- opérations

`len(`*s*`)`		*cardinal*
x **`in`** *s*		*x* ∈ *s*
s`.add(` *x* `), `*s*`.remove(` *x* `)`		*ajout et suppression d'élément*
s`.pop( )`		*extraction d'un élément*
s`.issubset(` *t* `)`	*s* `<=` *t*	*s* ⊂ *t*
s`.union(` *t* `)`	*s* `|` *t*	*s* ∪ *t*
s`.intersection(` *t* `)`	*s* `&` *t*	*s* ∩ *t*
s`.difference(` *t* `)`	*s* `-` *t*	
s`.copy()`		

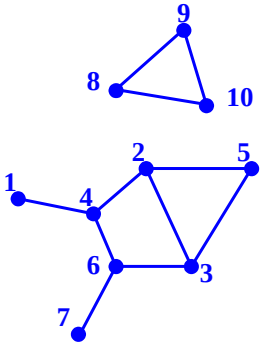
Set

- exemple : calcul de la composante connexe d'un sommet d'un graphe donné par la liste de ses arcs

```
gr = [(1, 4), (4, 1), (4, 2) … (8, 10)]  # graphe
x = 3                                     # sommet en question

compo = set( [ x ] )
stable = False
while not stable:
    stable = True
    for (orig, extr) in gr:
        if orig in compo and extr not in compo:
            compo.add(extr)
            stable = False

print(compo)
```



affichage obtenu :

```
set([1, 2, 3, 4, 5, 6, 7])
```


Dict

- ou *mémoire associative*, *table associative*, *map*, hash, etc.
- collection de paires (*clé* , *valeur*)
  la clé est le moyen de retrouver la valeur
- notation constructive :

  `{` *cle*$_1$ **`:`** *valeur*$_1$ **`,`** *cle*$_2$ **`:`** *valeur*$_2$ **`,`** **`…`** *cle*$_n$ **`:`** *valeur*$_n$ **`}`**

- principales opérations

dict `[` *clé* `]`	*renvoie la valeur associée à clé (si clé absente : ERREUR)*
dict`.get(` *clé*`,` *valdef* `)`	*renvoie la valeur associée à clé, valdef si clé absente*
dict `[` *clé* `] =` *valeur*	*définit (ou redéfinit) la valeur associée à clé*
clé **`in`** *dict*	*dict contient-il une paire avec la clé indiquée ?*
`del` *dict* `[` *clé* `]`	*supprime la clé et sa valeur associée*
dict`.items()`	*renvoie une copie de la liste des paires (clé, valeur)*
dict`.keys()`	*renvoie une copie de la liste des clés*
dict`.values()`	*renvoie une copie de la liste des valeurs*

Dict

- ex. : obtenir les mots d'un **`texte`** et le nombre d'occurrences de chacun

```
texte = "Ala Met Asn Glu Met Cys Asn Glu ... Glu Ile"
```

obtention de le liste des mots,
chacun avec le nombre de
ses apparitions dans le texte

- affichage :

```
Cys --> 2
Asp --> 6
Asn --> 3
```

etc.

Dict

- ex. : obtenir les mots d'un **`texte`** et le nombre d'occurrences de chacun

```
texte = "Ala Met Asn Glu Met Cys Asn Glu ... Glu Ile"

dict = {}
listeMots = texte.split()     # analyse lexicale (simpliste)
for mot in listeMots:
    if mot in dict:
        dict[mot] = dict[mot] + 1
    else:
        dict[mot] = 1

for c in res.keys():
    print(c, "-->", dict[c])
```

- affichage :

```
Cys --> 2
Asp --> 6
Asn --> 3
```

etc.

Dict

- Peut être construit à partir d'une liste de couples :

```
dict([(1, 2), (3, 4)]
```

- affichage :

```
{1: 2, 3: 4}
```