Éléments de Programmation
Cours 6 - Listes

Romain Demangeon

1I001 - Section 11 PEIP

11/10/2016

UPMC SORBONNE UNIVERSITÉS

Précisions

- Partiel: Mardi 26 Octobre, de 18h30 à 20h00.
- Tutorat de l'UE:
  - sans inscription (oiseau matinal),
  - Lundi 17/10 - 18h00 - A 421
  - Mardi 18/10 - 18h00 - A 421
  - Mercredi 19/10 - 18h00 - A 253
  - Jeudi 20/10 - 18h00 - A 261
- Supports: Partie 2 en cours de distribution.
- Aujourd'hui: notions difficiles (polymorphisme, mutabilité, méthodes, fonction internes).

Structures séquentielles

- Intervalles: séquences d'entiers consécutifs.
  - pratique pour itérer sur une suite d'entiers,
  - gestion mémoire optimisée.
- Chaînes de caractères: séquences de caractères (lettres et symboles)
  - pratique pour représenter des mots ou des phrases.
  - itération sur les caractères d'une chaîne.
  - accès direct à un caractère par son indice.
  - construction de chaînes par concaténation.
- Itérateur général pour les séquences: `for`.

Expressivité séquentielle limitée

- Séquence d'entiers non-consécutifs (Fibonacci, carrés) ?
- Séquence de données d'un même type arbitraire:
  - séquence de mots,
  - séquence de booléens,
  - séquence de séquences.

Structures séquentielles

Listes Homogènes

Organiser en séquence des éléments d'un même type.

- Ensemble ordonné:
  - itération avec `for`.
- Accès et construction directs,
  - comme pour les chaînes.
- Applications:
  - notes d'un étudiants (entiers),
  - données d'une expérience (couples $(x, f(x))$,
  - éléments d'une page web (objets).
  - tâches à effectuer (représentants).
  - notes d'une classe (couples (nom , liste d'entiers)).
- Problèmes sur les listes:
  - réduction,
  - transformation,
  - filtrage.

Types Polymorphes

On peut définir des fonctions (ou des expressions) qui calculent sur des objets informatiques peu importe leur type.
On utilise dans les signatures des variables de types (généralement α ou `alpha`) pour désigner un type quelconque.

```
def identite(x):
    """ alpha -> alpha """
    return x
```

- α (`alpha`) peut être n'importe quel type (`bool`, `int`, ...)
- `identite` est de type `alpha -> alpha`, mais pas `alpha -> beta`.

```
def mystere(x, y, z):
    """ alpha * beta * alpha -> bool """
    return x == z
```

Liste

Définition

Une liste de type `list[alpha]` est une séquence (finie) dont tous les éléments sont de type `alpha`.

- notes d'un étudiant: `list[int]`,
- mots d'une phrase: `list[str]`,
- notes d'une classe (Cours 07): `list[tuple[str, list[int]]]`.

Construction explicite

Une expression atomique de type `list[alpha]` est une séquence `[e_1, e_2, ..., e_n]` où chaque `e_i` est de type `alpha`.

- une séquence de 0 élément `[]` est appelée liste vide.

- Construction explicite avec crochet et virgule. (x)
- Les `e_i` ne sont pas forcéments des valeurs (expressions atomiques).

Listes: précisions

Typage

- la liste vide est de type `list[alpha]` pour tout `alpha`,
  - les listes vides de `list[int]` et de `list[bool]` sont le même objet informatique.
- Python est plus permissif que le cours, et autorise les listes hétérogènes:
  - `['aspirateur', 13, True]`
  - le cours de 1I001 reconnait uniquement l'existence des listes homogènes.
  - mauvais réflexe de programmation.

Longueur et Egalité

On dispose de primitives `len` et `==` qui calculent la longueur d'une liste et l'égalité de deux listes (comme pour les `str`).

- `len`: `list[alpha]` −> `int`
- `==`: `list[alpha]` * `list[alpha]` −> `bool`
- principe d'évaluation de `==`: égalité élément par élément.

Concaténation

On peut construire des listes par concaténation `+` de deux listes existantes.

- `+: list[alpha] * list[alpha] -> list[alpha]`
- `+` est associatif et pas commutatif.
- `[]` est l'élément neutre de `+`.
  - (`list[alpha]`, `+`, `[]`) est un monoïde.

- même `alpha` pour les deux paramètres de `+`:
  - listes homogènes.

Ajout d'élément (I)

Comme pour les `str`, on peut construire des listes éléments par éléments (par exemple en parcourant une autre liste).

- si `L: list[int]`, la signature de `+` nous interdit d'écrire `L + 3`,
- on doit écrire `L + [3]`,
- efficacité mémoire: on recopie la liste.

Mutabilité

Jusqu'ici

Les objets informatiques traités (`int`, `function`, `bool`, et surtout `str`) sont immutables.

- on ne peut pas les modifier,
- on peut remplacer une variable par une autre expression `s = s + c`
- pour supprimer un élément d'une chaîne, on reconstruit une nouvelle chaîne sans cet élément.

Mutabilité

Un objet informatique mutable peut être modifié: remplacement, suppression ou ajout d'une partie seulement de l'objet.

- les listes sont des objets mutables.

Méthodes vs. Fonctions

Fonctions

- Objets informatiques fonctionnels.
- Définition globale.
- Objet informatique indépendant: `f(x)`.
- Exécution dépendante uniquement des arguments.
- Plusieurs primitives (`min`, `math.sqrt`).

Méthodes

- Objets informatiques fonctionnels.
- Définition globale, comme attribut d'un autre objet.
- Objet informatique lié à un autre objet: `o_1.f(x)`.
- Exécution dépendante de l'objet lié (`o_1.f(3)` et `o_2.f(3)`).
- Plusieurs méthodes primitives pour objets de base (`append`, `reverse`)

Plus de précisions au Cours 10 (Objets).

La méthode d'objets mutables (listes) append

Définition

La méthode `append`, disponible sur toutes les listes, ajoute son argument à la fin de la liste depuis laquelle elle est appelée.

- si `Liste: list[alpha]`, alors `Liste.append: alpha -> NoneType`

- `append` est un effet de bord.
- différence entre `Liste.append(3)` et `Liste + [3]` ?
- `Liste.append` est une instruction.
- Principe d'interprétation de `Liste.append(e)`
  1. evaluation de `Liste` qui doit être un *objet liste* `L` de type `list[alpha]`,
  2. evaluation de `e` en `v` qui doit être de type `alpha`,
  3. ajout de `v` à la fin de l'objet liste `L`.
- Si l'objet liste est stocké dans une variable, le contenu de la variable est donc modifié.
- Que penser de `[2, 3].append(4)`.

Liste des entiers pairs

Problème

Donner une fonction `liste_pairs(n)` qui construit la liste des entiers pairs de $[1; n]$.

Liste des entiers pairs

Problème

Donner une fonction `liste_pairs(n)` qui construit la liste des entiers pairs de $[1; n]$.

```
def liste_pairs(n):
    """int -> list[int]
    Hypothese: 1 <= n

    retourne la liste des entiers pairs dans l'intervalle [1,n]."""

    # L : list[int]
    L = []  # la liste resultat, initialement vide

    # i : int (entier courant)
    for i in range(1, n + 1):
        if i % 2 == 0:
            L.append(i)  # ajout en fin, directement dans l
        # sinon : ne rien faire

    return L
```

- on aurait pu écrire `L = L + [i]`.
- efficacité moindre.

Liste des entiers pairs (Simulation)

Simulation de `liste_pairs(7)`

tour de boucle	variable i	variable L	Invariant
entrée	-	`[]`	
1	1	`[]`	
2	2	`[2]`	
3	3	`[2]`	
4	4	`[2, 4]`	
5	5	`[2, 4]`	
6	6	`[2, 4, 6]`	
7	7	`[2, 4, 6]`	
(sortie)	-	`[2, 4, 6]`	

Correction

- Invariant

Liste des entiers pairs (Simulation)

Simulation de `liste_pairs(7)`

tour de boucle	variable i	variable L	Invariant
entrée	-	`[]`	Vrai
1	1	`[]`	Vrai
2	2	`[2]`	Vrai
3	3	`[2]`	Vrai
4	4	`[2, 4]`	Vrai
5	5	`[2, 4]`	Vrai
6	6	`[2, 4, 6]`	Vrai
7	7	`[2, 4, 6]`	Vrai
(sortie)	-	`[2, 4, 6]`	Vrai

Correction

- Invariant "`L` contient la liste des entiers pairs de $[1, i]$" (existence `i`).
- Si invariant vrai en fin de boucle, le dernier élément de `range(1, n + 1)` est n, donc "`L` contient la liste des entiers pairs de $[1, n]$"

Terminaison

Liste des entiers pairs (Simulation)

Simulation de `liste_pairs(7)`

tour de boucle	variable i	variable L	Invariant
entrée	-	`[]`	Vrai
1	1	`[]`	Vrai
2	2	`[2]`	Vrai
3	3	`[2]`	Vrai
4	4	`[2, 4]`	Vrai
5	5	`[2, 4]`	Vrai
6	6	`[2, 4, 6]`	Vrai
7	7	`[2, 4, 6]`	Vrai
(sortie)	-	`[2, 4, 6]`	Vrai

Correction

- Invariant "`L` contient la liste des entiers pairs de $[1, i]$" (existence `i`).
- Si invariant vrai en fin de boucle, le dernier élément de `range(1, n + 1)` est n, donc "`L` contient la liste des entiers pairs de $[1, n]$"

Terminaison

Il suffit de dire que `range(1,n+1)` contient n éléments.

Accès direct à un élément

Syntaxe

On accède au $(i+1)$ème élément d'une liste `L` par `L[i]`.

- Si `L: list[alpha]`, alors `L[i]: alpha`

- Similaire aux chaînes de caractère.
- Différence: typage.
- indices commençant à `0`.

Exemple, liste `Fib = [1, 1, 2, 3, 5, 8, 13, 21, 34, 65]`

Elément	1	1	2	3	5	8	13	21	34	65
indice	0	1	2	3	4	5	6	7	8	9
indice inverse	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

- Utilisation possible de l'indice inverse.

Tranchage

Syntaxe

On accède à la sous-liste de la liste `L` qui commence à l'indice `i` (inclus) et finit à l'indice `j` (exclus) par `L[i:j]`.

- Si `Liste: list[alpha]` alors `Liste[i,j]: list[alpha]`

- Comparer `L[5]` et `L[5,6]`.
- Raccourcis:
  - `L[i:]` s'évalue en `L[i:len(L)]`,
  - `L[:j]` s'évalue en `L[0:j]`,
  - `L[:]` s'évalue en `L[0:len(L)]`, est-ce `L` ?
- Règles pour `L[i:j]`:
  - si `i>` `len(L)-1` ou si `j=0` alors `L[i:j]` renvoie la liste vide,
  - si `i<=` `len(L)-1` et si `j>` `len(L)` alors `L[i:j]` renvoie la même liste que `L[i:]`,
  - si `j<=` `len(L)-1` alors `L[i:-j]` renvoie la même liste que `L[i:len(L)-j]`,
  - si `j>` `len(L)-1` alors `L[i:-j]` renvoie la liste vide,
  - si `i<=` `len(L)` alors `L[-i:k]` renvoie la même liste que `L[len(L)-i:k]`.
  - si `i>` `len(L)` alors `L[-i:k]` renvoie la même liste que `L[0:k]`.

UPMC SORBONNE UNIVERSITÉS

Tranchage avec pas

Syntaxe: pas positif

On accède à la sous-liste de tous les éléments d'indices `i + n * k`, compris entre `i` (inclus) et `j` (exclus) de la liste `L`, `n` entier, avec `L[i:j:k]`.

- si `Liste: list[alpha]`, alors `L[i:j:k]: list[alpha]`

- par exemple `L[4:11:2]` renvoie la liste `[L[4], L[6], L[8], L[10]]`.
- `L[i:j:1]` est `L[i:j]`.

Syntaxe: pas négatif

On accède à la sous-liste de tous les éléments d'indices `j - n * k`, compris entre `j` (inclus) et `i` (exclus) de la liste `L`, `n` entier, avec `L[j:i:-k]`.

- si `Liste: list[alpha]`, alors `L[j:i:-k]: list[alpha]`

- par exemple `L[11:4:-2]` renvoie la liste `[L[11], L[9], L[7], L[5]]`.

Tranchage: exemple

Avec `A = ['Ses', 'ailes', 'de', 'geant', "l'", 'empeche', 'de', 'marcher']`

'Ses'	'ailes'	'de'	'geant'	"l'"	'empeche'	'de'	'marcher'
0	1	2	3	4	5	6	7
-8	-7	-6	-5	-4	-3	-2	-1

- `A[:4]`:

Tranchage: exemple

Avec `A = ['Ses', 'ailes', 'de', 'geant', "l'", 'empeche', 'de', 'marcher']`

'Ses'	'ailes'	'de'	'geant'	"l'"	'empeche'	'de'	'marcher'
0	1	2	3	4	5	6	7
-8	-7	-6	-5	-4	-3	-2	-1

- `A[:4]`: `['Ses', 'ailes', 'de', 'geant']`
- `A[4:]`:

Tranchage: exemple

Avec `A = ['Ses', 'ailes', 'de', 'geant', "l'", 'empeche', 'de', 'marcher']`

'Ses'	'ailes'	'de'	'geant'	"l'"	'empeche'	'de'	'marcher'
0	1	2	3	4	5	6	7
-8	-7	-6	-5	-4	-3	-2	-1

- `A[:4]`: `['Ses', 'ailes', 'de', 'geant']`
- `A[4:]`: `["l'", 'empeche', 'de', 'marcher']`
- `A[8:10]`:

Tranchage: exemple

Avec `A = ['Ses', 'ailes', 'de', 'geant', "l'", 'empeche', 'de', 'marcher']`

'Ses'	'ailes'	'de'	'geant'	"l'"	'empeche'	'de'	'marcher'
0	1	2	3	4	5	6	7
-8	-7	-6	-5	-4	-3	-2	-1

- `A[:4]`: `['Ses', 'ailes', 'de', 'geant']`
- `A[4:]`: `["l'", 'empeche', 'de', 'marcher']`
- `A[8:10]`: `[]`
- `A[4:23]`:

Tranchage: exemple

Avec `A = ['Ses', 'ailes', 'de', 'geant', "l'", 'empeche', 'de', 'marcher']`

'Ses'	'ailes'	'de'	'geant'	"l'"	'empeche'	'de'	'marcher'
0	1	2	3	4	5	6	7
-8	-7	-6	-5	-4	-3	-2	-1

- `A[:4]`: `['Ses', 'ailes', 'de', 'geant']`
- `A[4:]`: `["l'", 'empeche', 'de', 'marcher']`
- `A[8:10]`: `[]`
- `A[4:23]`: `["l'", 'empeche', 'de', 'marcher']`
- `A[-20:6]`:

Tranchage: exemple

Avec `A = ['Ses', 'ailes', 'de', 'geant', "l'", 'empeche', 'de', 'marcher']`

'Ses'	'ailes'	'de'	'geant'	"l'"	'empeche'	'de'	'marcher'
0	1	2	3	4	5	6	7
-8	-7	-6	-5	-4	-3	-2	-1

- `A[:4]`: `['Ses', 'ailes', 'de', 'geant']`
- `A[4:]`: `["l'", 'empeche', 'de', 'marcher']`
- `A[8:10]`: `[]`
- `A[4:23]`: `["l'", 'empeche', 'de', 'marcher']`
- `A[-20:6]`: `['Ses', 'ailes', 'de', 'geant', "l'", 'empeche']`
- `A[5:-5]`:

Tranchage: exemple

Avec `A = ['Ses', 'ailes', 'de', 'geant', "l'", 'empeche', 'de', 'marcher']`

'Ses'	'ailes'	'de'	'geant'	"l'"	'empeche'	'de'	'marcher'
0	1	2	3	4	5	6	7
-8	-7	-6	-5	-4	-3	-2	-1

- `A[:4]`: `['Ses', 'ailes', 'de', 'geant']`
- `A[4:]`: `["l'", 'empeche', 'de', 'marcher']`
- `A[8:10]`: `[]`
- `A[4:23]`: `["l'", 'empeche', 'de', 'marcher']`
- `A[-20:6]`: `['Ses', 'ailes', 'de', 'geant', "l'", 'empeche']`
- `A[5:-5]`: `[]`
- `A[1:8:2]`:

Tranchage: exemple

Avec `A = ['Ses', 'ailes', 'de', 'geant', "l'", 'empeche', 'de', 'marcher']`

'Ses'	'ailes'	'de'	'geant'	"l'"	'empeche'	'de'	'marcher'
0	1	2	3	4	5	6	7
-8	-7	-6	-5	-4	-3	-2	-1

- `A[:4]`: `['Ses', 'ailes', 'de', 'geant']`
- `A[4:]`: `["l'", 'empeche', 'de', 'marcher']`
- `A[8:10]`: `[]`
- `A[4:23]`: `["l'", 'empeche', 'de', 'marcher']`
- `A[-20:6]`: `['Ses', 'ailes', 'de', 'geant', "l'", 'empeche']`
- `A[5:-5]`: `[]`
- `A[1:8:2]`: `['ailes', 'geant', 'empeche', 'marcher']`
- `A[1::2]`:

Tranchage: exemple

Avec `A = ['Ses', 'ailes', 'de', 'geant', "l'", 'empeche', 'de', 'marcher']`

'Ses'	'ailes'	'de'	'geant'	"l'"	'empeche'	'de'	'marcher'
0	1	2	3	4	5	6	7
-8	-7	-6	-5	-4	-3	-2	-1

- `A[:4]`: `['Ses', 'ailes', 'de', 'geant']`
- `A[4:]`: `["l'", 'empeche', 'de', 'marcher']`
- `A[8:10]`: `[]`
- `A[4:23]`: `["l'", 'empeche', 'de', 'marcher']`
- `A[-20:6]`: `['Ses', 'ailes', 'de', 'geant', "l'", 'empeche']`
- `A[5:-5]`: `[]`
- `A[1:8:2]`: `['ailes', 'geant', 'empeche', 'marcher']`
- `A[1::2]`: `['ailes', 'geant', 'empeche', 'marcher']`
- `A[7:0:-1]`:

Tranchage: exemple

Avec `A = ['Ses', 'ailes', 'de', 'geant', "l'", 'empeche', 'de', 'marcher']`

'Ses'	'ailes'	'de'	'geant'	"l'"	'empeche'	'de'	'marcher'
0	1	2	3	4	5	6	7
-8	-7	-6	-5	-4	-3	-2	-1

- `A[:4]`: `['Ses', 'ailes', 'de', 'geant']`
- `A[4:]`: `["l'", 'empeche', 'de', 'marcher']`
- `A[8:10]`: `[]`
- `A[4:23]`: `["l'", 'empeche', 'de', 'marcher']`
- `A[-20:6]`: `['Ses', 'ailes', 'de', 'geant', "l'", 'empeche']`
- `A[5:-5]`: `[]`
- `A[1:8:2]`: `['ailes', 'geant', 'empeche', 'marcher']`
- `A[1::2]`: `['ailes', 'geant', 'empeche', 'marcher']`
- `A[7:0:-1]`: `['marcher', 'de', 'empeche', "l'", 'geant', 'de', 'ailes']`
- avec le premier élément ?

Tranchage: exemple

Avec `A = ['Ses', 'ailes', 'de', 'geant', "l'", 'empeche', 'de', 'marcher']`

'Ses'	'ailes'	'de'	'geant'	"l'"	'empeche'	'de'	'marcher'
0	1	2	3	4	5	6	7
-8	-7	-6	-5	-4	-3	-2	-1

- `A[:4]`: `['Ses', 'ailes', 'de', 'geant']`
- `A[4:]`: `["l'", 'empeche', 'de', 'marcher']`
- `A[8:10]`: `[]`
- `A[4:23]`: `["l'", 'empeche', 'de', 'marcher']`
- `A[-20:6]`: `['Ses', 'ailes', 'de', 'geant', "l'", 'empeche']`
- `A[5:-5]`: `[]`
- `A[1:8:2]`: `['ailes', 'geant', 'empeche', 'marcher']`
- `A[1::2]`: `['ailes', 'geant', 'empeche', 'marcher']`
- `A[7:0:-1]`: `['marcher', 'de', 'empeche', "l'", 'geant', 'de', 'ailes']`
- avec le premier élément ? `A[7:-9:-1]`

Itération sur une séquence: principe d'interprétation

Pour interpréter:

```
for var in seq:
    instr_1
    instr_2
    ...
    instr_n
instr_s
```

1. On évalue `seq`
2. Si `seq` est vide on quitte la boucle et interprète `instr_s`
3. Sinon, on crée une nouvelle variable `var` initialisée au premier élément de `seq`
4. On interprète les instructions `instr_i` du corps dans l'ordre.
   - elle peuvent utiliser `var`.
5. Si il y a un élément suivant `var` dans `seq`:
   5.1 on met l'élément suivant dans `var`.
   5.2 on revient en 4.
6. Sinon:
   - on détruit `var`.
   - on sort de l'itération et on interprète `instr_s`

Itération sur une liste: principe d'interprétation

Pour interpréter:

```
for e in L:
    instr_1
    instr_2
    ...
    instr_n
instr_s
```

1. On évalue `L`
2. Si `L` est `[]` on quitte la boucle et interprète `instr_s`
3. Sinon, on crée une nouvelle variable `e` initialisée à `L[0]`,
4. On interprète les instructions `instr_i` du corps dans l'ordre.
   - elle peuvent utiliser `e`.
5. Si il y a un élément suivant `e` (`L[i]`) dans `L`:
   - 5.1 on met l'élément suivant `L[i+1]` dans `e`.
   - 5.2 on revient en 4.
6. Sinon:
   - on détruit `e`.
   - on sort de l'itération et on interprète `instr_s`

Problèmes sur les listes

- Réduction (par exemple `list[int] -> bool` ou `list[alpha] -> int`)
  - on compte quelque chose dans la liste,
  - on vérifie une propriété dans la liste,
  - on extrait un élément de la liste, ...
- Transformation (par exemple `list[alpha] -> list[beta]`)
  - on modifie certains (ou tous les) éléments de la liste,
  - on calcule quelque chose pour chaque élément, ...
- Filtrage (par exemple `list[alpha] -> list[alpha]`)
  - on supprime certains éléments d'une liste en fonction de propriétés.
- Autres problèmes:
  - construire une liste à partir de deux listes,
  - ajouter des éléments dans une liste,
  - ...

Attention

Les listes sont mutables, mais on ne modifie jamais les paramètres d'une fonction.

- on n'écrit pas de fonctions/méthodes qui modifient une liste,
- on écrit uniquement des fonctions qui renvoient une (autre) liste.

Réduction: Somme des éléments d'une liste

Problème

Donner une fonction `somme_liste(L)` qui retourne la somme des éléments de `L`.

- signature: `list[Number] -> Number`
- `somme_liste([2, 10, 0, 4])` donne `16`.
- `somme_liste(liste_pairs(6))` donne `12`.

Réduction: Somme des éléments d'une liste

Problème

Donner une fonction `somme_liste(L)` qui retourne la somme des éléments de `L`.

- signature: `list[Number] -> Number`
- `somme_liste([2, 10, 0, 4])` donne `16`.
- `somme_liste(liste_pairs(6))` donne `12`.

```
def somme_liste(L):
    """list[Number] -> Number

    Retourne la somme des elements de la liste L."""

    # s : Number
    s = 0  # la somme cummulee des elements

    # n : Number  (element courant)
    for n in L:
        s = s + n

    return s
```

Réduction: Somme des éléments d'une liste (Simulation)

```python
def somme_liste(L):
    """list[Number] -> Number

    Retourne la somme des elements de la liste L."""

    # s : Number
    s = 0  # la somme cummulee des elements

    # n : Number  (element courant)
    for n in L:
        s = s + n

    return s
```

Simulation de `sommei_liste(L_1)`

Réduction: Somme des éléments d'une liste (Simulation)

```python
def somme_liste(L):
    """list[Number] -> Number

    Retourne la somme des elements de la liste L."""

    # s : Number
    s = 0  # la somme cummulee des elements

    # n : Number  (element courant)
    for n in L:
        s = s + n

    return s
```

Simulation de `sommei_liste(L_1)`

Avec `L_1 = [4, 2, 1, 3, 5]`

tour de boucle	variable n	variable s
entrée	-	0
1	4	4
2	2	6
3	1	7
4	3	10
5	5	15
(sortie)	-	15

Réduction: Longueur d'une liste

Problème

Donner une fonction `longueur(L)` qui retourne la longueur d'une liste `L`.

- signature: `Liste[alpha] -> int`
- `longueur([2,10,0,4])` donne `4`.
- `longueur(['granule'])` donne `1`.
- `longueur([])` donne `0`.
- `longueur(liste_pairs(6))` donne `3`.

Réduction: Longueur d'une liste

Problème

Donner une fonction `longueur(L)` qui retourne la longueur d'une liste `L`.

- signature: `Liste[alpha] -> int`
- `longueur([2,10,0,4])` donne `4`.
- `longueur(['granule'])` donne `1`.
- `longueur([])` donne `0`.
- `longueur(liste_pairs(6))` donne `3`.

```
def longueur(L):
    """list[alpha] -> int

    Retourne la longueur de la liste L."""

    # lng : int
    lng = 0  # comptage de la longueur initialement a zero

    # e : alpha   (element courant)
    for e in L:
        lng = lng + 1  # longueur incrementee pour chaque element

    return lng
```

- Fonction polymorphe: instanciation de type.

Transformation: `map`

Schéma

Le schéma de transformation (schéma map) applique une même fonction à tous les éléments d'une liste:

- on dispose de:
  1. une liste `L` de type `list[alpha]`
  2. une fonction `f` de type `alpha -> beta`
- on veut construire map `f L`, c.-a.-d.:
  - la liste `[f(L[0]), f(L[1]), ..., f(L[-1])]`
  - de type `list[beta]`

- ubiquitaire en informatique,
- paradigme parallèle *"MapReduce"*.
  - calculer le maximum d'argent dépensé par un client sur une année.

Transformation: Liste des carrés

Problème

Donner une fonction `liste_carres(L)` qui renvoie la liste des carrés des éléments de la liste `L`.

- on construit une fonction `carre: Number -> Number`,
- signature: `liste_carres: list[Number] -> list[Number]`
- schéma map `carre L`
- `liste_carres([4, 1, 2, 2])` donne `[16, 1, 4, 4]`.

Transformation: Liste des carrés

Problème

Donner une fonction `liste_carres(L)` qui renvoie la liste des carrés des éléments de la liste `L`.

- on construit une fonction `carre: Number -> Number`,
- signature: `liste_carres: list[Number] -> list[Number]`
- schéma map `carre L`
- `liste_carres([4, 1, 2, 2])` donne `[16, 1, 4, 4]`.

```
def carre(x):
    """Number -> Number
    Retourne le nombre x eleve au carre."""

    return x * x

def liste_carres(L):
    """list[Number] -> list[Number]
    Retourne la liste des elements de L eleves au carre."""

    # LR : list[Number]
    LR = []  # liste resultat, initialement vide

    # n : Number (element courant)
    for x in L:
        LR.append(carre(x)) # ajoute le carre en fin de resultat

    return LR
```

Transformation: Liste des carrés (Simulation)

```
def liste_carres(L):
    """list[Number] -> list[Number]
    Retourne la liste des elements de L eleves au carre."""

    # LR : list[Number]
    LR = []  # liste resultat, initialement vide

    # n : Number (element courant)
    for x in L:
        LR.append(carre(x)) # ajoute le carre en fin de resultat

    return LR
```

Simulation `liste_carres([4, 1, 2, 2])`

tour de boucle	variable x	variable LR
entrée	-	[]
1	4	[16]
2	1	[16, 1]
3	2	[16, 1, 4]
4	2	[16, 1, 4, 4]
(sortie)	-	[16, 1, 4, 4]

Transformation: Liste des longueurs de chaînes

Problème

Donner une fonction `liste_longueurs_chaines(L)` qui renvoie la liste des longueurs des chaînes de caractères éléments de la liste `L`.

- on utilise une fonction `len: str -> int`,
- signature: `liste_longueurs_chaines: list[str] -> list[int]`
- schéma `map len L`
- avec `A = ['Ses', 'ailes', 'de', 'geant', "l'", 'empeche', 'de', 'marcher']`, `liste_longueurs_chaines(A)` donne `[3, 5, 2, 5, 2, 7, 2, 7]`.

Transformation: Liste des longueurs de chaînes

Problème

Donner une fonction `liste_longueurs_chaines(L)` qui renvoie la liste des longueurs des chaînes de caractères éléments de la liste `L`.

- on utilise une fonction `len: str -> int`,
- signature: `liste_longueurs_chaines: list[str] -> list[int]`
- schéma map `len L`
- avec `A = ['Ses', 'ailes', 'de', 'geant', "l'", 'empeche', 'de', 'marcher']`, `liste_longueurs_chaines(A)` donne `[3, 5, 2, 5, 2, 7, 2, 7]`.

```
def liste_longueurs_chaines(L):
    """list[str] -> list[int]
    retourne la liste des longueurs des chaines elements de L."""

    # LR : list[int]
    LR = []  # liste resultat, initialement vide

    # s : str (element courant)
    for s in L:
        LR.append(len(s)) # ajoute la longueur en fin de resultat

    return LR
```

Filtrage: `filter`

Schéma

Le schéma de filtrage (schéma filter) applique une prédicat à tous les éléments d'une liste et conserve uniquement ceux validant le prédicat:

- on dispose de:
  1. une liste `L` de type `list[alpha]`
  2. une fonction `p` de type `alpha -> bool`
- on veut construire filter `p L`, c.-a.-d.:
  - la liste `[L[i_0], L[i_1], ..., L[i_k]]`
  - telle que `i_0` est l'indice du premier élément de `L` pour lequel on a `p(L[i_0]) == True`, ...
  - de type `list[alpha]`

- commun en informatique,
  - extraire la liste des étudiants ayant validé leur année.

Filtrage: Liste des entiers pairs

Problème

Donner une fonction `liste_des_pairs(L)` qui renvoie la liste des éléments de la liste `L` qui sont pairs.

- on construit une fonction `est_pair: int -> bool`,
- signature: `liste_des_pairs: list[int] -> list[int]`
- schéma filter `est_pair L`
- `liste_des_pairs([1, 4, 5, 6, 2, 3])` donne `[4, 6, 2]`

Filtrage: Liste des entiers pairs

Problème

Donner une fonction `liste_des_pairs(L)` qui renvoie la liste des éléments de la liste `L` qui sont pairs.

- on construit une fonction `est_pair: int -> bool`,
- signature: `liste_des_pairs: list[int] -> list[int]`
- schéma filter `est_pair L`
- `liste_des_pairs([1, 4, 5, 6, 2, 3])` donne `[4, 6, 2]`

```
def est_pair(n):
    """int -> bool
    renvoie True si l'entier n est pair, False sinon."""

    return n % 2 == 0

def liste_des_pairs(L):
    """list[int] -> list[int]
    Retourne la liste des entiers pairs elements de L."""

    # LR : list[int]
    LR = []  # la liste filtree, initialement vide

    # n : int (element courant)
    for n in L:
        if est_pair(n):
            LR.append(n)
        # sinon on ne fait rien

    return LR
```

Filtrage: Liste des entiers pairs (Simulation)

```python
def liste_des_pairs(L):
    """list[int] -> list[int]
    Retourne la liste des entiers pairs elements de L."""

    # LR : list[int]
    LR = []  # la liste filtree, initialement vide

    # n : int (element courant)
    for n in L:
        if est_pair(n):
            LR.append(n)
        # sinon on ne fait rien

    return LR
```

Simulation `liste_des_pairs([1, 4, 5, 6, 2, 3])`

tour de boucle	variable n	variable LR
entrée	-	`[]`
1	1	`[]`
2	4	`[4]`
3	5	`[4]`
4	6	`[4, 6]`
5	2	`[4, 6, 2]`
6	3	`[4, 6, 2]`
(sortie)	-	`[4, 6, 2]`

Filtrage: Liste des entiers supérieurs à un nombre

Problème

Donner une fonction `liste_superieurs(L, x)` qui renvoie la liste des éléments de la liste `L` qui sont supérieurs strictement à `x`.

- on utilise une "fonction" $>$ `x: int -> bool`,
- signature: `liste_superieurs: list[int] * int -> list[int]`
- schéma filter $(\texttt{y} \mapsto \texttt{y} > \texttt{x})$ `L`
- `liste_superieurs([1, 4, 5, 6, 2, 3], 3)` donne `[4, 5, 6]`

Filtrage: Liste des entiers supérieurs à un nombre

Problème

Donner une fonction `liste_superieurs(L, x)` qui renvoie la liste des éléments de la liste `L` qui sont supérieurs strictement à `x`.

- on utilise une "fonction" `> x: int -> bool`,
- signature: `liste_superieurs: list[int] * int -> list[int]`
- schéma filter $(\texttt{y} \mapsto \texttt{y} > \texttt{x})$ `L`
- `liste_superieurs([1, 4, 5, 6, 2, 3], 3)` donne `[4, 5, 6]`

```
def liste_superieurs(L, x):
    """list[Number] * Number -> list[Number]
    renvoie la liste des nombres elements de L superieurs
    au nombre x."""

    # LR : list[Number]
    LR = []  # la liste resultat initialement vide

    # y : Number (element nombre courant)
    for y in L:
        if y > x:
            LR.append(y)
        # sinon ne rien faire

    return LR
```

Liste des entiers supérieurs à un nombre (II)

- on utilise une fonction interne `est_superieur: int -> bool`,
- signature: `liste_superieurs: list[int] * int -> list[int]`
- schéma filter `est_superieur L`

```
def liste_superieurs(L, x):
    """list[Number] * Number -> list[Number]

    renvoie la liste des nombres elements de L superieurs
    au nombre x."""

    # Voici la definition interne :
    def superieur_a_x(z):
        """Number -> bool

        Renvoie True si z est superieur a x, False sinon."""
        return z > x

    # et maintenant le reste du corps de la fonction principale

    # LR : list[Number]
    LR = []  # la liste resultat initialement vide

    # y : Number (element nombre courant)
    for y in L:
        if superieur_a_x(y):
            LR.append(y)
        # sinon ne rien faire

    return LR
```

- portée de la fonction interne.

Efficacité (Rappel)

Définition

Un programme `f` est *plus efficace dans le pire des cas* qu'un programme `g`:

- `f` et `g` calculent la même fonction mathématique.
- sur toutes les entrées d'une même taille, dans le pire des cas, `f` utilise moins d'opérations élémentaires que `g`.

```
def mention(m):
    """ int -> str
    hypothese: m >= 0 and m <= 20
    renvoie la mention du bac associee a la moyenne m"""

    if m <= 10:
        return ''Elimine''
    else:
        if m <= 12:
            return ''Passable''
        else:
            if m <= 14:
                return ''Assez Bien''
            else:
                if m <= 16:
                    return ''Bien''
                else:
                    return ''Tres bien''
```

```
def mention(m):
    """ int -> str
    hypothese: m >= 0 and m <= 20
    renvoie la mention du bac associee a la moyenne m"""

    if m <= 14:
        if m <= 12:
            if m <= 10:
                return ''Elimine''
            else:
                return ''Passable''
        else:
            return ''Assez Bien''
    else:
        if m <= 16:
            return ''Bien''
        else:
            return ''Tres bien''
```

Complexité exacte

Définition

La complexité de la fonction `f` est la fonction $\mathcal{C} : \mathbb{N} \to \mathbb{N}$ qui à un entier n, associe $\mathcal{C}(n)$ le nombre maximum d'opérations élémentaires que fait la fonction `f` lors d'un calcul sur un argument de taille n.

- opérations élémentaires: il faut choisir ce que l'on compte (comparaisons, multiplications, affectations, ...),
- entrées de taille n: il faut pouvoir définir une "taille" des entrées (longueur d'une liste, ...),
- nombre maximum: complexité "dans le pire des cas".

Complexité exacte (exemple)

```
def presence(e, L):
    """ List[alpha] * alpha -> bool"""

    # trouve : bool
    trouve = False

    # c : int
    c = 0

    while (not trouve) and (c < len(L)):
        if e == L[c]:
            trouve = True
        else:
            c = c + 1
    return trouve
```

- opérations élémentaires:

Complexité exacte (exemple)

```
def presence(e, L):
    """ List[alpha] * alpha -> bool"""

    # trouve : bool
    trouve = False

    # c : int
    c = 0

    while (not trouve) and (c < len(L)):
        if e == L[c]:
            trouve = True
        else:
            c = c + 1
    return trouve
```

- opérations élémentaires: comparaisons d'éléments ==,
- entrées de taille n:

Complexité exacte (exemple)

```python
def presence(e, L):
    """ List[alpha] * alpha -> bool"""

    # trouve : bool
    trouve = False

    # c : int
    c = 0

    while (not trouve) and (c < len(L)):
        if e == L[c]:
            trouve = True
        else:
            c = c + 1
    return trouve
```

- opérations élémentaires: comparaisons d'éléments `==`,
- entrées de taille n: longueur de `L`,
- nombre maximum:

Complexité exacte (exemple)

```
def presence(e, L):
    """ List[alpha] * alpha -> bool"""

    # trouve : bool
    trouve = False

    # c : int
    c = 0

    while (not trouve) and (c < len(L)):
        if e == L[c]:
            trouve = True
        else:
            c = c + 1
    return trouve
```

- opérations élémentaires: comparaisons d'éléments `==`,
- entrées de taille n: longueur de `L`,
- nombre maximum: pire des cas, `e` pas dans `L`.
- $\mathcal{C}(n)$ vaut "le nombre maximum de comparaisons `==` que `presence` fait sur tous les couples `e`, `L` où `L` est de taille n.

$\mathcal{C}(0) = 0, \mathcal{C}(1) = 1, \ldots, \mathcal{C}(n) = n$

Complexité exacte (exemple II)

```
def presence_couple(L):
    """ List[alpha] -> bool"""

    # i : int
    for i in range(0, len(L)):
        # j : int
        for j in range(0, len(L)):
            if (i != j) and (L[i] == L[j]):
                return True
    return False
```

- opérations élémentaires:

Complexité exacte (exemple II)

```python
def presence_couple(L):
    """ List[alpha] -> bool"""

    # i : int
    for i in range(0, len(L)):
        # j : int
        for j in range(0, len(L)):
            if (i != j) and (L[i] == L[j]):
                return True
    return False
```

- opérations élémentaires: comparaisons d'éléments ==,
- entrées de taille n:

Complexité exacte (exemple II)

```python
def presence_couple(L):
    """ List[alpha] -> bool"""

    # i : int
    for i in range(0, len(L)):
        # j : int
        for j in range(0, len(L)):
            if (i != j) and (L[i] == L[j]):
                return True
    return False
```

- opérations élémentaires: comparaisons d'éléments `==`,
- entrées de taille n: longueur de `L`,
- nombre maximum:

Complexité exacte (exemple II)

```python
def presence_couple(L):
    """ List[alpha] -> bool"""

    # i : int
    for i in range(0, len(L)):
        # j : int
        for j in range(0, len(L)):
            if (i != j) and (L[i] == L[j]):
                return True
    return False
```

- opérations élémentaires: comparaisons d'éléments ==,
- entrées de taille n: longueur de L,
- nombre maximum: pire des cas, pas de couples dans L.
- $\mathcal{C}(n)$ vaut "le nombre maximum de comparaisons == que presence_couple fait sur toutes les listes L de taille n".

$\mathcal{C}(0) = 0$, $\mathcal{C}(1) = 0$, $\mathcal{C}(2) = 2$, $\mathcal{C}(3) = 6$, $\ldots$, $\mathcal{C}(n) = n * (n-1)$

Complexité exacte (exemple II)

```python
def presence_couple(L):
    """ List[alpha] -> bool"""

    # i : int
    for i in range(0, len(L)):
        # j : int
        for j in range(0, len(L)):
            if (i != j) and (L[i] == L[j]):
                return True
    return False
```

- opérations élémentaires: comparaisons d'éléments ==,
- entrées de taille n: longueur de L,
- nombre maximum: pire des cas, pas de couples dans L.
- $\mathcal{C}(n)$ vaut "le nombre maximum de comparaisons == que presence_couple fait sur toutes les listes L de taille n".

$\mathcal{C}(0) = 0$, $\mathcal{C}(1) = 0$, $\mathcal{C}(2) = 2$, $\mathcal{C}(3) = 6$, $\ldots$, $\mathcal{C}(n) = n * (n-1)$

- Peut-on faire mieux ?

Complexité exacte (exemple III)

```
def anagrammes(S):
    """ str -> List[str]"""
    T = ['']
    for i in range(0, len(S)):
        R = []
        for s in T:
            for j in range(0, i + 1):
                R.append(s[:j] + S[i] + s[j:])
        T = R
    return R
```

- opérations élémentaires:

Complexité exacte (exemple III)

```
def anagrammes(S):
    """ str -> List[str]"""
    T = ['']
    for i in range(0, len(S)):
        R = []
        for s in T:
            for j in range(0, i + 1):
                R.append(s[:j] + S[i] + s[j:])
        T = R
    return R
```

- opérations élémentaires: modifications de liste `append`,
- entrées de taille n:

Complexité exacte (exemple III)

```
def anagrammes(S):
    """ str -> List[str]"""
    T = ['']
    for i in range(0, len(S)):
        R = []
        for s in T:
            for j in range(0, i + 1):
                R.append(s[:j] + S[i] + s[j:])
        T = R
    return R
```

- opérations élémentaires: modifications de liste `append`,
- entrées de taille n: longueur de `L`,
- nombre maximum:

Complexité exacte (exemple III)

```python
def anagrammes(S):
    """ str -> List[str]"""
    T = ['']
    for i in range(0, len(S)):
        R = []
        for s in T:
            for j in range(0, i + 1):
                R.append(s[:j] + S[i] + s[j:])
        T = R
    return R
```

- opérations élémentaires: modifications de liste `append`,
- entrées de taille n: longueur de `L`,
- nombre maximum: toutes les entrées de taille n sont similaires.
- $\mathcal{C}(n)$ vaut "le nombre de modification de liste que `anagrammes` fait sur une chaine de longueur n".

$\mathcal{C}(0) = 0$, $\mathcal{C}(1) = 1$, $\mathcal{C}(2) = 2$, $\mathcal{C}(3) = 6$, $\ldots$, $\mathcal{C}(n) \approx n!$

Complexité Asymptotique

Définition

Soient f et g deux fonctions de $\mathbb{N} \to \mathbb{N}$.
On note $f(n) = \Theta(g(n))$, "f domine et est soumise par g" quand il existe deux constantes k_1 et k_2 et un rang n_0 tels que:

- $\forall n \geq n_0,\ k_1.g(n) \leq f(n) \leq k_2.g(n)$

- $f(n) = \Theta(g(n))$ quand "f croît comme g".
- on a $\frac{n.(n-1)}{2} = \Theta(n^2)$
- on a $106 * n + 2000 = \Theta(n)$
- Cela permet de définir des classes de complexités:
  - classe linéaire: $\mathcal{C}(n) = \Theta(n)$, par exemple `presence`
  - classe quadratique: $\mathcal{C}(n) = \Theta(n^2)$, par exemple `presence_couple`
  - classe factorielle: $\mathcal{C}(n) = \Theta(n!)$, par exemple `anagrammes`

Comparaison de complexité

- Problème: Insertion dans une liste déjà triée.
- `insertion(3, [1, 2, 4, 5])` donne `[1, 2, 3, 4, 5]`.

```
def insert(e, L):
    R = []
    i = 0
    while (L[i] < e) and (i < len(L)):
        R.append(L[i])
        i = i + 1
    R.append(e)
    while (i < len(L)):
        R.append(L[i])
        i = i + 1
    return R
```

$\mathcal{C}(n) = \Theta(n)$

complexité linéaire.

```
def insert(e, L):
    g = 0
    d = len(L) - 1
    while g != d:
        m = (g + d) // 2
        if e < m:
            d = m
        else:
            g = m
    return L[:g] + e + L[g:]
```

$\mathcal{C}(n) = \Theta(\log_2(n))$

complexité logarithmique.

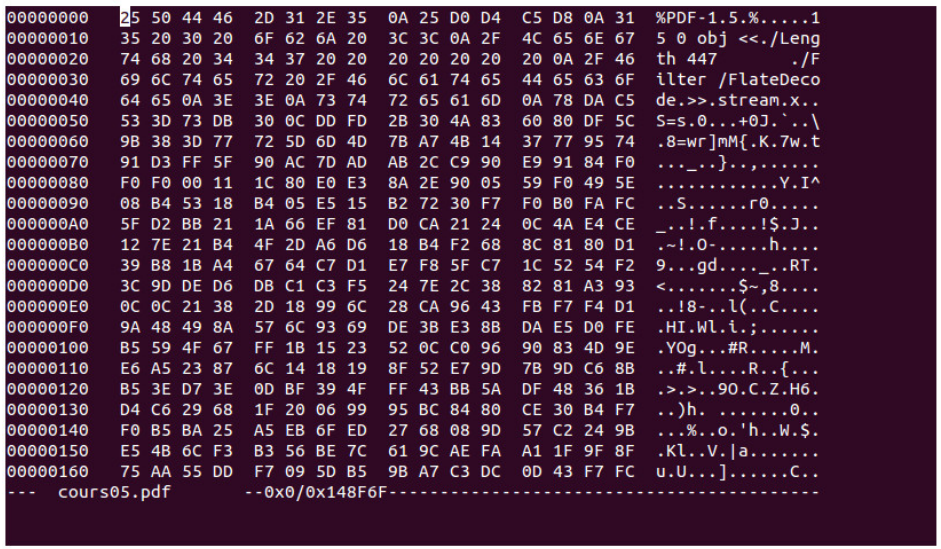
- deux fonctions de complexités asymptotiques différentes qui résolvent le même problème.
- si une comparaison prend 0.001s, sur une liste de taille 1 000 000, la première fonctions peut prendre 1/4 h alors que la deuxième 0.02s.

Fichiers

- Les données sont organisées dans la mémoire en fichiers.
- Le Système d'Exploitation (Windows, Unix, IOS, Android, ...) s'occupe (entre autres) de la gestion des fichiers.
- Un fichier est ciblé par:
  - un chemin de dossiers `/home/romain/upmc/enseignement/`
    - moyen pour le s.e. de trouver rapidement le fichier.
  - un nom `cours05`
    - moyen pour le s.e. de distinguer les fichier du même chemin.
  - une extension `.pdf`
    - moyen pour le s.e. de choisir l'application à utiliser.

Fichiers (II)

- Le format du fichier est la manière dont les données sont codées et organisée.
  - brique de base, l'octet (*byte*) (huit 0 ou 1, nombre entre 0 et 255),
  - l'en-tête du fichier donne le format et d'autres information (métadonnées, date de création),
  - le corps du fichier contient les données, encodées selon le format.
- Le fichier est une séquence d'octets consécutifs de la mémoire,
  - elle est lue dans une seule direction.
  - pdf du cours (début):



- On peut écrire des chaînes de caractères dans un fichier.
- On peut utiliser Python pour manipuler des fichiers.

Ouverture de fichier

Syntaxe

On ouvre un fichier en python avec `open` de type `str * str -> file`.
Dans `open(nom, droit)`

- `nom` est le nom du fichier,
- `droit` est une chaine indiquant les droits d'acces au fichier.
  - exemple: `'w'` ou `'r+'`,

- `'w'` donne le droit d'écriture.
- `'r'` donne le droit de lecture.
- `'r+'` et `'w+'` donne les deux.
- quelques subtilités (c.f. référence du langage).
- par exemple, `f = open("cours05-15.tex","r")`

On doit toujours fermer un fichier que l'on a ouvert avec la méthode `close` du fichier.

Lecture d'un fichier

Syntaxe

On lit un fichier ouvert avec la méthode `read` et `readline` du fichier de type `file -> str`.

- On lit généralement un fichier ligne par ligne (dans l'ordre).
  - caractère de fin de ligne `'\n'`.

```
def extrait(titre, n1, n2):
    """ str * Number ** 2 -> str
    retourne le texte du fichier titre.txt
    compris entre les lignes n1 et n2"""

    #i :int
    i = 0
    #s: str
    s = ""
    #corbeille: str
    corbeille = ""
    #f: file
    f = open(titre + ".txt", "r")

    while i < n2:
        if i < n1:
            corbeille = f.readline()
        else:
            s = s + f.readline()
        i = i +1
    f.close()
    return s
```

Ecriture d'un fichier

Syntaxe

On écrit dans un fichier ouvert avec la méthode `write` du fichier de type `file` –> NoneType.

```
def premiers(n):
    """ Number -> NoneType
    ecrit dans un fichier les nombres premiers inferieur a n"""

    #p:int
    p = 2

    #s:str
    s = ""

    #f: file
    f = open("premiers.txt", "w+")

    while p <= n:
        s = "Le nombre " + str(p)
        if est_premier(p):
            s = s + " est premier.\n"
        else:
            s = s + " n'est pas premier.\n"
        f.write(s)
        p = p + 1
    f.close()

    return s
```

Conclusion

- Notions fondamentales d'informatique:
  - polymorphisme,
  - mutabilité,
  - méthode/fonction,
  - schémas de calcul,
  - fonctions internes.
- Construction, accès, itération avec des listes.
- Types de problèmes: réduction, transformation, filtrage.

Cours 07

Ces *n*-uplets qui nous gouvernent. Résolution de problèmes.