Éléments de Programmation + Cours 04 - Itération

Romain Demangeon

LU1IN011 - Section ScFo 11 + 13 + ...

03/10/2022

Correction et boucles

```
def puissance(x : float, n : int) -> float:
    """ Precondition : n>=0
    retourne la valeur de x eleve a la puissance n."""

    res : float = 1 # valeur de x^0

    i : int = 1 # compteur

    while i != n + 1:
        res = res * x
        i = i + 1
    return res
```

Simulation `puissance(2,5)`

Correction et boucles

```
def puissance(x : float, n : int) -> float:
    """ Precondition : n>=0
    retourne la valeur de x eleve a la puissance n."""

    res : float = 1 # valeur de x^0

    i : int = 1 # compteur

    while i != n + 1:
        res = res * x
        i = i + 1
    return res
```

Simulation `puissance(2,5)`

tour de boucle	variable res	variable i
entrée	1	1
1	2	2
2	4	3
3	8	4
4	16	5
5 (sortie)	32	6

Correction et boucles

```python
def puissance(x : float, n : int) -> float:
    """ Precondition : n>=0
    retourne la valeur de x eleve a la puissance n."""

    res : float = 1 # valeur de x^0

    i : int = 1 # compteur

    while i != n + 1:
        res = res * x
        i = i + 1
    return res
```

Simulation `puissance(2,5)`

tour de boucle	variable res	variable i
entrée	1	1
1	2	2
2	4	3
3	8	4
4	16	5
5 (sortie)	32	6

La fonction est-elle correcte ?

Invariant

Malaise

- la simulation nous prouve formellement que `puissance` est correcte quand elle est appelée sur les arguments `2, 5`.
- la simulation ne nous dit rien, formellement, sur le cas général.
- la simulation nous suggère informellement une méthode générale:
  - à chaque étape, on voit qu'on multiplie `res` par `x`,
  - on s'arrête après `n` étapes.

Définition

Un *invariant de boucle* est une expression logique:

- Qui est vraie en entrée de boucle.
- Qui est vraie après chaque tour de boucle.

- on ne dit rien de l'invariant au cours du calcul d'une boucle.
- on veut des invariants utiles: pas `i >= 1` ni `True`.
- l'invariant est une expression logique (pas `Python`)

Invariant (II)

Qu'est ce qu'un invariant utile ?

- il faut que "(Invariant) + (Sortie de boucle) $\Rightarrow$ Correction"

Méthode Standard

1. Comprendre le problème posé.
   - exprimer la correction.
2. Simuler la fonction (plusieurs fois).
   - intuition de ce qui reste vrai à chaque tour de boucle.
   - relation entre les variables et arguments.
3. Expérience et vision mathématique.
   - penser à ce qui entraîne la correction.

Invariant pour `puissance`

Invariant (II)

Qu'est ce qu'un invariant utile ?

- il faut que "(Invariant) + (Sortie de boucle) $\Rightarrow$ Correction"

Méthode Standard

1. Comprendre le problème posé.
   - exprimer la correction.
2. Simuler la fonction (plusieurs fois).
   - intuition de ce qui reste vrai à chaque tour de boucle.
   - relation entre les variables et arguments.
3. Expérience et vision mathématique.
   - penser à ce qui entraîne la correction.

Invariant pour `puissance`

$$\texttt{x}^{\texttt{i}-1} = \texttt{res}$$

Invariant (III)

Simulation avec invariant de `puissance(2, 5)`

Invariant (III)

Simulation avec invariant de `puissance(2, 5)`

tour de boucle	variable res	variable i	Invariant $x^{i-1} = res$
entree	1	1	$2^{1-1} = 1$**(Vrai)**
1	2	2	$2^{2-1} = 2$ **(Vrai)**
2	4	3	$2^{3-1} = 4$ **(Vrai)**
3	8	4	$2^{4-1} = 8$ **(Vrai)**
4	16	5	$2^{5-1} = 16$ **(Vrai)**
5 (sortie)	32	6	$2^{6-1} = 32$ **(Vrai)**

Cas général

On veut montrer que l'invariant reste vrai à chaque tour de boucle:

- Preuve par récurrence:
  - Invariant vrai en entrée.
  - On suppose que l'invariant est vrai au début d'un tour de boucle, on montre qu'il est vrai à la fin du tour de boucle.
  - par récurrence, l'invariant est vrai en sortie de boucle.

Invariant (IV) - Preuve Formelle

Montrons, par récurrence, que $\texttt{x}^{\texttt{i}-1} = \texttt{res}$ est toujours vrai en sortie de boucle:

- On a $\texttt{x}^{1} - {}^{1} = 1$, donc l'invariant est vrai en entrée de boucle.
- On appelle x, n les valeurs de `x` et `n`, on appelle i, r les valeurs de `i` et `res` au début du tour et i', r' leurs valeurs en fin de tour.
  - Supposons que l'invariant est vrai au début d'un tour. On a $x^{i-1} = r$.
  - en regardantle corps de la boucle, on sait que:
    - $i' = i + 1$,
    - $r' = r * x$,
  - on calcule $r' = r * x = x^{i-1} * x = x^{i} = x^{i'-1}$
  - et l'invariant est vrai à la fin du tour.
- par récurrence, l'invariant est toujours vrai en fin de tour, donc en sortie de boucle (si jamais ça arrive, cf. terminaison).

On a montré que notre invariant est invariant. Est-il utile ?

Invariant (V) - Preuve Formelle

Montrons que "(Invariant) + (Sortie de boucle) $\Rightarrow$ Correction"

- En sortie de boucle, l'invariant est vrai (cf. slide précédent): donc $x^{i-1} = res$ (1)
- En sortie de boucle, la condition de boucle est fausse (par définition): donc $i = n + 1$ (2)
- De (1) et (2) on déduit: $x^n = res$
- La fonction renvoie la valeur de `res`, donc elle est correcte (pour la fonction mathématique "puissance").

Aux Examens de LU1IN001

- Recherche d'invariant: pas vraiment (QCM).
- Preuve d'invariance: non (L2 Info).
  - Test de l'invariant dans une simulation.
- Preuve de correction en sortie de boucle: oui.

Terminaison

- La méthode de correction suppose que la fonction termine (que la sortie de boucle existe).
- Comment prouver qu'une fonction avec un `while` termine ?
  - Pas de preuve générale (indécidabilité).
  - Idée: montrer que quelque chose décroît strictement à chaque tour.
  - Corollaire de *Bolzano-Weierstrass* (caractérisation des espaces métriques compacts): il n'existe pas de suite infinie d'entiers naturels strictement décroissante.

Définition

Un *variant de boucle* est une expression arithmétique:

- Qui est un entier naturel positif en entrée de boucle.
- Qui décroit **strictement** à chaque tour de boucle.
- Qui vaut 0 en sortie de boucle.

Trouver un bon variant:

- même méthode que pour l'invariant,
- intuition de ce qui décroît.

Variant

Variant pour `puissance`:

Variant

Variant pour `puissance:n` - `i` + 1

Simulation de `puissance(2, 5)` avec variant

Variant

Variant pour `puissance:n - i` $+ 1$

Simulation de `puissance(2, 5)` avec variant

tour de boucle	variable res	variable i	Variant `n - i` $+ 1$
entree	1	1	$5 - 1 + 1 = 5$
1	2	2	$5 - 2 + 1 = 4$
2	4	3	$5 - 3 + 1 = 3$
3	8	4	$5 - 4 + 1 = 2$
4	16	5	$5 - 5 + 1 = 1$
5 (sortie)	32	6	$5 - 6 + 1 = 0$

Cas général

On veut montrer qu'une expression est un variant:

- Montrer que sa valeur est un entier positif en entrée de boucle.
- Montrer que sa valeur décroit strictement entre le début et la fin d'un tour de boucle.
- Montrer qu'on sort de la boucle quand il vaut 0.

Variant (II) - Preuve formelle

- Montrons que $\texttt{n} - \texttt{i} + 1$ est bien un variant de boucle:
  - Appelons n la valeur de `n` et i_0 la valeur de `i` en entrée de boucle.
  - En entrée l'expression $n - i_0 + 1 = n - 1 + 1$ vaut n qui est un entier positif.
  - Appelons i la valeur de `i` au début d'un tour et i' la valeur de `i` à la fin d'un tour.
    - on sait que $i' = i + 1$,
    - donc $n - i' + 1 = n - (i + 1) + 1 = n - i < n - i + 1$
    - le variant décroit scrictement à chaque tour de boucle.
  - quand le variant vaut 0, on a $n - i + 1 = 0$ soit $i = n + 1$ ce qui correspond à la condition de sortie de boucle.
- Comme la fonction admet un variant de boucle, elle termine.

Aux Examens de LU1IN001

- Recherche de variant: oui.
- Preuve de terminaison: non (simulation).
  - Test du variant sur une simulation.

Efficacité

Points à examiner en LU1IN001

- Calculs redondants.
- Raccourcis logiques.
- Algorithme plus efficace.

Contexte

- Conjecture empirique de Moore: *la densité des transistors double tous les deux ans*.
  - croissance exponentielle de la puissance de calcul
- Taille des données croît aussi (graphismes).
- Méthodes basées sur la non-efficacité des programmes: cryptographie.
- Classes de complexité des fonctions mathématiques: $\mathcal{P}$, $\mathcal{NP}$, ...
  - $f \in \mathcal{P}$ si il existe un programme `f` et un polynôme P tel que pour tout x `f(x)` calcule $f(x)$ en temps $t \leq P(|x|)$.

Factorisation

```python
def aire_triangle(a : float, b : float, c : float) -> float:
    """ Precondition : (a>0) and (b>0) and (c>0)
    Precondition : les cotes a, b, et c definissent
    bien un triangle.

    donne l'aire du triangle dont les cotes
    sont de longueur a, b, et c."""

    return math.sqrt(((a + b + c) / 2)
                     * (((a + b + c) / 2) - a)
                     * (((a + b + c) / 2) - b)
                     * (((a + b + c) / 2) - c))
```

```python
def aire_triangle(a : float, b : float, c : float) -> float:
    """ Precondition : (a>0) and (b>0) and (c>0)
    Precondition : les cotes a, b, et c definissent
    bien un triangle.

    retourne l'aire du triangle dont les cotes
    sont de longueur a, b, et c."""

    p : float = (a + b + c) / 2   # demi-perimetre

    return math.sqrt(p * (p - a) * (p - b) * (p - c))
```

- Calcul `(a + b + c) / 2` répété 4 fois.
- Factorisation du calcul grâce à une variable.
- Complexité en espace / Coût d'une affectation.

Sortie anticipée (I)

On veut calculer le plus petit diviseur non-trivial (différent de 1) d'un entier naturel.

```
def plus_petit_diviseur(n : int) -> int:
    """ retourne le plus petit diviseur non-trivial de n"""

    d : int = 0 # pas encore trouve

    m : int = 2

    while m <= n:
        if (d == 0) and (n % m == 0):
            d = m
        m = m + 1
     return d
```

Sortie anticipée (I)

On veut calculer le plus petit diviseur non-trivial (différent de 1) d'un entier naturel.

```
def plus_petit_diviseur(n : int) -> int:
    """ retourne le plus petit diviseur non-trivial de n"""

    d : int = 0 # pas encore trouve

    m : int = 2

    while m <= n:
        if (d == 0) and (n % m == 0):
            d = m
        m = m + 1
    return d
```

```
def plus_petit_diviseur(n : int) -> int:
    """ retourne le plus petit diviseur non-trivial de
    d : int = 0 # pas encore trouve

    m : int = 2

    while (d == 0) and (m <= n):
        if (d == 0) and (n % m == 0):
            d = m
        m = m + 1
    return d
```

- On évite (quand n n'est pas premier) beaucoup de tours de boucle inutiles.
- Raffinement de la condition de la boucle.

Sortie anticipée (II)

```
def plus_petit_diviseur(n : int) -> int:
    """ retourne le plus petit diviseur non-trivial de n"""

    d : int = 0 # pas encore trouve

    m : int = 2

    while m <= n:
        if (d == 0) and (n % m == 0):
            return m
        m = m + 1
    return d
```

- Sortie directe de la fonction (avec return).
- `return` fait sortir de la fonction, donc *a fortiori* de toutes les boucles.
- Analyse de terminaison ?

Compter l'efficacité

- Définir ce que l'on compte:
  - dépend du contexte:
    - avion:

Compter l'efficacité

- Définir ce que l'on compte:
  - dépend du contexte:
    - avion: nombre d'opérations numériques.
    - micro-onde:

Compter l'efficacité

- Définir ce que l'on compte:
  - dépend du contexte:
    - avion: nombre d'opérations numériques.
    - micro-onde: espace mémoire.
    - jeux vidéo:

Compter l'efficacité

- Définir ce que l'on compte:
  - dépend du contexte:
    - avion: nombre d'opérations numériques.
    - micro-onde: espace mémoire.
    - jeux vidéo: appels aux fonctions de la la librairie graphique.
    - appli web:

Compter l'efficacité

- Définir ce que l'on compte:
  - dépend du contexte:
    - avion: nombre d'opérations numériques.
    - micro-onde: espace mémoire.
    - jeux vidéo: appels aux fonctions de la la librairie graphique.
    - appli web: envoi et réception de messages asynchrones (AJAX).
  - temps:
    - multiplications, comparaisons, appels à des primitives, ...
  - espace:
    - nombre de variables, appels de fonctions.
- Definir par rapport à quoi on compte:
  - dans quel cas ?
    - en moyenne, pire des cas, ...
  - taille des arguments:
    - longueur d'une liste,
    - taille d'un entier (son $\texttt{log}_2$, correspondant à la mémoire qu'il prend)
- On s'intéresse souvent à la complexité asymptotique:
  - $\frac{n.(n+1)}{2}$ est similaire à $3.n^2 + 180.n$.
  - $n.\texttt{log}_2(n)$ est meilleur que $\frac{n.(n+1)}{2}$.

Efficacité d'algorithmes

```python
def puissance(x : float, n : int) -> float:
    """ Precondition : n >= 0
    retourne x eleve a la puissance n."""

    res : float = 1

    i : int = 1

    while i != (n + 1):
        res = res * x
        i = i + 1
    return res
```

- Autant de multiplications que la valeur `n`
- Peut-on faire mieux ?

Efficacité d'algorithmes

```
def puissance(x : float, n : int) -> float:
    """ Precondition : n >= 0
    retourne x eleve a la puissance n."""

    res : float = 1

    i : int = 1

    while i != (n + 1):
        res = res * x
        i = i + 1
    return res
```

- Autant de multiplications que la valeur `n`
- Peut-on faire mieux ?

- $x^n = \begin{cases} x^{\lfloor n/2 \rfloor} * x^{\lfloor n/2 \rfloor} & \text{si } n \text{ est pair} \\ x^{\lfloor n/2 \rfloor} * x^{\lfloor n/2 \rfloor} * x & \text{sinon} \end{cases}$

```
def puissance_rapide(x : float, n ; int) -> float:
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    res : float = 1

    acc : float = x

    i : int = n

    while i > 0:
        if i % 2 == 1:
            res = res * acc
        acc = acc * acc
        i = i // 2
    return res
```

- Est-ce que ça calcule vraiment x^n ?

Efficacité d'algorithmes

```
def puissance(x : float, n : int) -> float:
    """ Precondition : n >= 0
    retourne x eleve a la puissance n."""

    res : float = 1

    i : int = 1

    while i != (n + 1):
        res = res * x
        i = i + 1
    return res
```

- Autant de multiplications que la valeur `n`
- Peut-on faire mieux ?

- $x^n = \begin{cases} x^{\lfloor n/2 \rfloor} * x^{\lfloor n/2 \rfloor} & \text{si } n \text{ est pair} \\ x^{\lfloor n/2 \rfloor} * x^{\lfloor n/2 \rfloor} * x & \text{sinon} \end{cases}$

```
def puissance_rapide(x : float, n ; int) -> float:
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    res : float = 1

    acc : float = x

    i : int = n

    while i > 0:
        if i % 2 == 1:
            res = res * acc
        acc = acc * acc
        i = i // 2
    return res
```

- Est-ce que ça calcule vraiment x^n ?
- Preuve de correction

Puissance rapide - Terminaison

Variant ?

Puissance rapide - Terminaison

Variant `i`.

Simulation `puissance_rapide(2,10)` avec variant

Puissance rapide - Terminaison

Variant `i`.

Simulation `puissance_rapide(2,10)` avec variant

tour de boucle	variable res	variable acc	Variant `i`
entree	1	2	10
1	1	4	5
2	4	16	2
3	4	256	1
4 (sortie)	1024	65536	0

Preuve Formelle

- i_0 valeur initiale de `i` vaut n valeur de `n`.
- i valeur de `i` en début de boucle, i' valeur en fin de boucle:
  - $i' = \lfloor i/2 \rfloor < i$
- quand $i = 0$ on sort de la boucle (condition fausse)

Ainsi, `puissance_rapide` termine.

Puissance rapide - Correction

Invariant ?

Puissance rapide - Correction

Invariant $\texttt{res} = \frac{x^n}{acc^i}$.

Puissance rapide - Correction

Invariant $\texttt{res} = \frac{x^n}{acc^i}$.

tour de boucle	variable res	variable acc	variable i	Invariant $\texttt{res} = \frac{x^n}{acc^i}$
entree	1	2	10	$1 = \frac{1024}{2^{10}}$ **(Vrai)**
1	1	4	5	$1 = \frac{1024}{4^5}$ **(Vrai)**
2	4	16	2	$4 = \frac{1024}{16^2}$ **(Vrai)**
3	4	256	1	$4 = \frac{1024}{256^1}$ **(Vrai)**
4 (sortie)	1024	65536	0	$1024 = \frac{1024}{65536^0}$ **(Vrai)**

- Soit n, x les valeurs de `n`, `x`, invariant vrai en entrée: $1 = \frac{x^n}{x^n}$.
- Soit i, a, r les valeurs de `i`, `acc`, `res` en début de tour et i', a', r' leurs valeurs en fin de tour.
  - Supposons l'invariant vrai en début de tour: $r = \frac{x^n}{a^i}$
  - Si $i = 2 * p$, alors $i' = p$, $r' = r$, $a' = a * a$, on a $r' = \frac{x^n}{a^{2p}} = \frac{x^n}{(a*a)^p} = \frac{x^n}{a'^{i'}}$ et l'invariant reste vrai.
  - Si $i = 2 * p + 1$, alors $i' = p$, $r' = r * a$, $a' = a * a$, on a $r' = a * \frac{x^n}{a^{2p+1}} = a * \frac{x^n}{(a*a)^p * a} = \frac{x^n}{a'^{i'}}$ et l'invariant reste vrai.
- Par récurrence, invariant vrai en sortie, quand $\texttt{i} = 0$ soit $\texttt{res} = \frac{x^n}{\text{acc}^0}$
  La fonction renvoie bien x^n.

Complexité

```
def puissance(x : float, n : int) -> float:
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    res : Number = 1

    i : int = 1

    while i != (n + 1):
        res = res * x
        i = i + 1
    return res
```

- Autant de multiplications que la valeur `n`.
- On fait une de multiplication à chaque tour de boucle.
- On fait n tours de boucle.
- Dans tous les cas, $n \times 1 = n$ multiplications.

Complexité

```
def puissance(x : float, n : int) -> float:
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    res : Number = 1

    i : int = 1

    while i != (n + 1):
        res = res * x
        i = i + 1
    return res
```

- Autant de multiplications que la valeur `n`.
- On fait une de multiplication à chaque tour de boucle.
- On fait n tours de boucle.
- Dans tous les cas, $n \times 1 = n$ multiplications.

```
def puissance_rapide(x : float, n : int):
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    # res : Number
    res = 1

    # val : Number
    acc = x

    # i : int
    i = n

    while i > 0:
        if i % 2 == 1:
            res = res * acc
        acc = acc * acc
        i = i // 2
    return res
```

- Combien de multiplications ?

Complexité

```
def puissance(x : float, n : int) -> float:
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    res : Number = 1

    i : int = 1

    while i != (n + 1):
        res = res * x
        i = i + 1
    return res
```

- Autant de multiplications que la valeur `n`.
- On fait une de multiplication à chaque tour de boucle.
- On fait n tours de boucle.
- Dans tous les cas, $n \times 1 = n$ multiplications.

```
def puissance_rapide(x : float, n : int):
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    # res : Number
    res = 1

    # val : Number
    acc = x

    # i : int
    i = n

    while i > 0:
        if i % 2 == 1:
            res = res * acc
        acc = acc * acc
        i = i // 2
    return res
```

- Combien de multiplications ?
- On fait une ou deux multiplications à chaque tour de boucle.
- On fait

Complexité

```
def puissance(x : float, n : int) -> float:
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    res : Number = 1

    i : int = 1

    while i != (n + 1):
        res = res * x
        i = i + 1
    return res
```

- Autant de multiplications que la valeur `n`.
- On fait une de multiplication à chaque tour de boucle.
- On fait n tours de boucle.
- Dans tous les cas, $n \times 1 = n$ multiplications.

```
def puissance_rapide(x : float, n : int):
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    # res : Number
    res = 1

    # val : Number
    acc = x

    # i : int
    i = n

    while i > 0:
        if i % 2 == 1:
            res = res * acc
        acc = acc * acc
        i = i // 2
    return res
```

- Combien de multiplications ?
- On fait une ou deux multiplications à chaque tour de boucle.
- On fait $\log_2(n)$ tours de boucle.
- Dans le pire des cas, $2.\log_2(n)$ multiplications.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Tracer la complexité

- On peut stocker le nombre d'opérations élémentaires que fait une fonction dans une variable.
  - et l'afficher avec `print` juste avant la sortie de la fonction.

```python
def puissance_complex(x : float, n : int):
    """ Precondition : n>= 0 """

    res : float = 1

    i : int = 1

    nb_mult : int = 0

    while i <= n:
        res = res * x
        nb_mult = nb_mult + 1
        i = i + 1

    print("Nombre de multiplications =", nb_mult)

    return res
```

```python
def puissance_rapide_complex(x : float, n : int) -> int:
    """ Precondition : n >= 0 """

    res : float = 1

    acc : float = x

    i : int = n

    nb_mult : int = 0

    while i > 0:
        if i % 2 == 1:
            res = res * acc
            nb_mult = nb_mult + 1

        acc = acc * acc
        nb_mult = nb_mult + 1
        i = i // 2

    print("Nombre de multiplications = ",nb_mult)
    return res
```

Récursion

- Autre approche (que la boucle) pour les calculs répétitifs.
- Fonction qui s'utilise elle-même.
- Dans le corps d'une fonction `f` on peut appliquer une fonction `g`
  - exemple: `divise` dans `est_premier`
  - et quand `g` = `f` ?

Récursion

- Autre approche (que la boucle) pour les calculs répétitifs.
- Fonction qui s'utilise elle-même.
- Dans le corps d'une fonction `f` on peut appliquer une fonction `g`
  - exemple: `divise` dans `est_premier`
  - et quand `g` = `f` ?

```
def factorielle(n : int) -> int:
    """ Precondition : n >= 0
    donne la factorielle de n."""

    if n <= 1:
        return 1
    else:
        return n * factorielle(n - 1)
```

- Proche de la définition mathématique.
- Correction ? Terminaison ?
- Pas particulièrement efficace (récursivité terminale).
- Pas au programme de LU1IN001 (cf. LU2IN019, LI101).

Exercice: Racine Cubique

```python
def racine_cubique_entiere(n : int) -> int:
    """ Precondition : n >= 0 """

    racine : int  = 0
    while (racine ** 3) <= n:
        racine = racine + 1
    return racine - 1
```

- Variant ?

Exercice: Racine Cubique

```python
def racine_cubique_entiere(n : int) -> int:
    """ Precondition : n >= 0 """

    racine : int  = 0
    while (racine ** 3) <= n:
        racine = racine + 1
    return racine - 1
```

- Variant $\texttt{max}(n - racine^3, 0)$,
- Invariant ?

Exercice: Racine Cubique

```python
def racine_cubique_entiere(n : int) -> int:
    """ Precondition : n >= 0 """

    racine : int  = 0
    while (racine ** 3) <= n:
        racine = racine + 1
    return racine - 1
```

- Variant $\texttt{max}(n - racine^3, 0)$,
- Invariant "$(racine - 1)^3 \leq n$",
- Correction ?

Exercice: Racine Cubique

```python
def racine_cubique_entiere(n : int) -> int:
    """ Precondition : n >= 0 """

    racine : int  = 0
    while (racine ** 3) <= n:
        racine = racine + 1
    return racine - 1
```

- Variant $\texttt{max}(n - racine^3, 0)$,
- Invariant "$(racine - 1)^3 \leq n$",
- Correction en fin de boucle, $racine^3 > n$ donc $(racine - 1)^3 \leq n < racine^3$.
- Complexité (en comparaison, pire des cas) ?

Exercice: Racine Cubique

```python
def racine_cubique_entiere(n : int) -> int:
    """ Precondition : n >= 0 """

    racine : int  = 0
    while (racine ** 3) <= n:
        racine = racine + 1
    return racine - 1
```

- Variant $\texttt{max}(n - racine^3, 0)$,
- Invariant "$(racine - 1)^3 \leq n$",
- Correction en fin de boucle, $racine^3 > n$ donc $(racine - 1)^3 \leq n < racine^3$.
- Complexité (en comparaison, pire des cas)
  - pire des cas: indifférent.
  - complexité: $n^{\frac{1}{3}}$

Exercice: PGCD

```python
def pgcd(n : int, m : int) -> int:
    """ Precondition : n >= m>= 0 """

    d : int = n

    r : int = m

    # temp : int
    temp = 0

    while r != 0:
        temp = d % r
        d = r
        r = temp

    return d
```

- Variant ?

Exercice: PGCD

```
def pgcd(n : int, m : int) -> int:
    """ Precondition : n >= m>= 0 """

    d : int = n

    r : int = m

    # temp : int
    temp = 0

    while r != 0:
        temp = d % r
        d = r
        r = temp

    return d
```

- Variant r,
- Invariant ?

Exercice: PGCD

```
def pgcd(n : int, m : int) -> int:
    """ Precondition : n >= m>= 0 """

    d : int = n

    r : int = m

    # temp : int
    temp = 0

    while r != 0:
        temp = d % r
        d = r
        r = temp

    return d
```

- Variant r,
- Invariant "$\mathtt{div(n,m)} = \mathtt{div(d,r)}$",
- Correction ?

Exercice: PGCD

```python
def pgcd(n : int, m : int) -> int:
    """ Precondition : n >= m>= 0 """

    d : int = n

    r : int = m

    # temp : int
    temp = 0

    while r != 0:
        temp = d % r
        d = r
        r = temp

    return d
```

- Variant r,
- Invariant "$\mathtt{div}(\mathtt{n},\mathtt{m}) = \mathtt{div}(\mathtt{d},\mathtt{r})$",
- Correction en fin de boucle, $r = 0$ donc
  $\mathtt{div}(\mathtt{n},\mathtt{m}) = \mathtt{div}(\mathtt{d},0) = \mathtt{div}(\mathtt{d})$, puis passage au $\mathtt{max}$.
- Complexité (en modulo, pire des cas) ?

Exercice: PGCD

```python
def pgcd(n : int, m : int) -> int:
    """ Precondition : n >= m>= 0 """

    d : int = n

    r : int = m

    # temp : int
    temp = 0

    while r != 0:
        temp = d % r
        d = r
        r = temp

    return d
```

- Variant r,
- Invariant "$\mathtt{div}(\mathtt{n},\mathtt{m}) = \mathtt{div}(\mathtt{d},\mathtt{r})$",
- Correction en fin de boucle, $r = 0$ donc
  $\mathtt{div}(\mathtt{n},\mathtt{m}) = \mathtt{div}(\mathtt{d},0) = \mathtt{div}(\mathtt{d})$, puis passage au $\mathtt{max}$.
- Complexité (en modulo, pire des cas)
  - pire des cas: nombres consécutifs de Fibonacci.
  - complexité: inférieure à $k.\log_2(m) + 1$ avec $k \approx 2.0781$

Correction de programmes

- Domaine Systèmes Embarqués: comment être sûr que l'avion ne va pas s'écraser ? (ou que le missile va s'écraser)
  - Le problème de la correction d'un programme est indécidable.
  - Deux approches:
    - le Test: soumettre le programme à des jeux de test couvrant.
    - la Vérification: regarder le code du programme, prouver la correction.
  - Avantages
    - Test: peu coûteux, pas d'accès au code.
    - Vérification: à faire une seule fois, garantie formelle.
  - Inconvénients
    - Test: pas de garantie (cas improbables mais possibles ?).
    - Vérification: très difficile.
  - En pratique: beaucoup de test, mais de plus en plus de vérification.

Vérification

- Systèmes de Types: des types garantissent certaines propriétés: correction, terminaison, absence de blocage, ...
  - exemple: typeur de *MrPython*
- Modèles: considérer l'espace d'état (les configurations possibles de la mémoire) et le diviser en zone.
  - montrer que certaines zones sont inatteignables.

Vérification automatique

- Des assistants de preuves (Coq, Isabelle) peuvent prouver des programmes.
- Code basé sur les isomorphismes de Curry-Howard.

  formules $\leftrightarrow$ types
  preuves $\leftrightarrow$ programmes
- Utile pour vérifier les interpréteurs et compilateurs des langages de programmation.

Modèles de Calcul

Définition

Un modèle de calcul est un langage formel de termes liés par une relation de réduction.

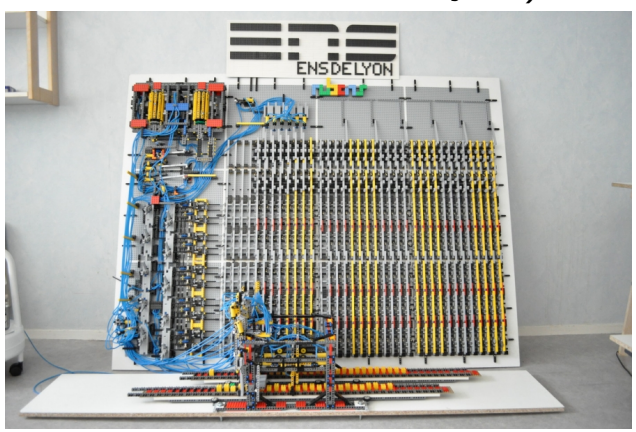
- syntaxe: langage de termes, construit depuis des briques de base (Python 101: instructions et expressions, "arbres" du Cours01)
- sémantique: règles de réduction (Python 101: principes d'interprétation et d'évaluation)

Exemples de modèles de calcul

- Fonctions Récursives (Kleene 1940):
  - fonctions des entiers définies récursivement.
  - syntaxe: définition, successeur, projection, composition, récursion, minimisation.
  - exemple: $\mathtt{Add}(a, b) = \mathbf{R}^2[\mathbf{U}_1{}^1, \mathbf{S}_1{}^3(\mathtt{Succ}, \mathbf{U}_2{}^3)]$
  - sémantique: remplacement en utilisant des égalités.

Modèles de Calcul

- Lambda-Calcul (Church 1930):
  - tout est fonction, calcul = substitution de variables.
  - syntaxe: $M, N ::= x \mid \lambda x.M \mid M\ N$.
  - exemple: $\texttt{Add}(a, b) = \lambda abfe.(a\ f)\ (b\ f\ e)$
  - sémantique: β-reduction: $(\lambda x.M)\ N \to M[N/x]$
  - origine du `lambda` de Python.
- Machine de Turing (Turing 1930):
  - modèle opérationnel de calcul: "ordinateur".
  - syntaxe: un ruban (de 0 et de 1), une tête de lecture, un état.
  - sémantique: règles, en fonction de la position de la tête et de la case du ruban, on modifie (ou non) la case et on se déplace à gauche ou à droite
  - en Lego (par les étudiants de l'ENS de Lyon)



Trois manières différentes de calculer.

Thèse de Church

(Rappel) Expressivité: ensembles des fonctions mathématiques atteignables par un modèle de calcul.

Théorème

Les machines de Turing, le λ-calcul de Church et les fonctions récursives de Kleene:

1. ont la même expressivité,
2. qui est la notion naturelle du calcul.

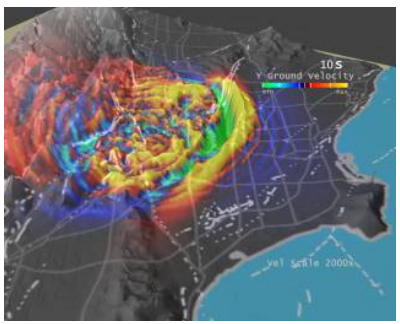
- partie mathématique (1.): preuves formelles, encodages.
- partie philosophique (2.): notion de "calcul du cerveau humain".
- introduction de la Turing-complétude (Turing-puissance): "être aussi expressif que ces 3 modèles".
  - les langages usuels sont (évidemment) Turing-puissant.
  - notre langage (à quatre instructions) est Turing-puissant.

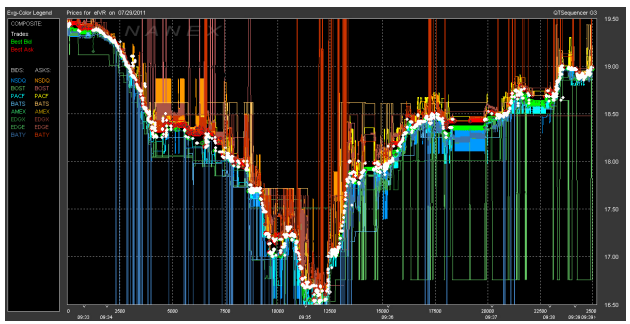
Expressivité

Etat du langage

- Langage suffisamment expressif (Turing-complet):
  - expressions booléennes et numériques.
  - instructions de boucles, alternative, affectation, retour.
- Calculs numériques et logiques complexes.
- Notions de correction, terminaison, efficacité.

- Genèse des ordinateurs: calculs numériques (*Pascaline*, *ENIAC*).
- Aujourd'hui, calculs numériques:
  - Recherche (modèles et simulations)
  - Animation / Jeux vidéo (calculs d'image)
  - Finance (modèles, prédictions)





Que des entiers.

Représentation

- On peut tout représenter avec des entiers:
  - Codages de Gödel: existence de fonctions bijectives de $\mathbb{N}^2 \to \mathbb{N}$, de $\mathbb{N}^m \to \mathbb{N}$, et surtout de $\mathbb{N}^* \to \mathbb{N}$.
  - Analogie avec la machine:
    - codages par bits (*binary digits*) de l'information.
- Tout calcul devient fonction des entiers.
  - on représente un argument par un entier:
    - texte: codage des lettres,
    - image: codage de la couleurs des pixels,
    - base de données: codage des champs, ...
  - les fonctions qu'on manipule opèrent sur des entiers (représentant des données argument) et renvoie un entier (représentant des données résultat).

Que des entiers.

Représentation

- On peut tout représenter avec des entiers:
  - Codages de Gödel: existence de fonctions bijectives de $\mathbb{N}^2 \to \mathbb{N}$, de $\mathbb{N}^m \to \mathbb{N}$, et surtout de $\mathbb{N}^* \to \mathbb{N}$.
  - Analogie avec la machine:
    - codages par bits (*binary digits*) de l'information.
- Tout calcul devient fonction des entiers.
  - on représente un argument par un entier:
    - texte: codage des lettres,
    - image: codage de la couleurs des pixels,
    - base de données: codage des champs, ...
  - les fonctions qu'on manipule opèrent sur des entiers (représentant des données argument) et renvoie un entier (représentant des données résultat).

Est-ce raisonnable ?

Que des entiers.

Représentation

- On peut tout représenter avec des entiers:
  - Codages de Gödel: existence de fonctions bijectives de $\mathbb{N}^2 \to \mathbb{N}$, de $\mathbb{N}^m \to \mathbb{N}$, et surtout de $\mathbb{N}^* \to \mathbb{N}$.
  - Analogie avec la machine:
    - codages par bits (*binary digits*) de l'information.
- Tout calcul devient fonction des entiers.
  - on représente un argument par un entier:
    - texte: codage des lettres,
    - image: codage de la couleurs des pixels,
    - base de données: codage des champs, ...
  - les fonctions qu'on manipule opèrent sur des entiers (représentant des données argument) et renvoie un entier (représentant des données résultat).

Est-ce raisonnable ?

- du point de vue de la machine: peut-être,

Que des entiers.

Représentation

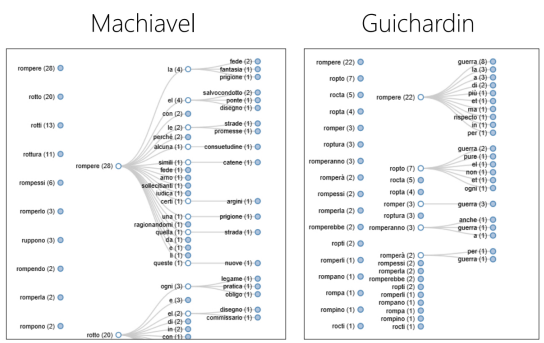
- On peut tout représenter avec des entiers:
  - Codages de Gödel: existence de fonctions bijectives de $\mathbb{N}^2 \to \mathbb{N}$, de $\mathbb{N}^m \to \mathbb{N}$, et surtout de $\mathbb{N}^* \to \mathbb{N}$.
  - Analogie avec la machine:
    - codages par bits (*binary digits*) de l'information.
- Tout calcul devient fonction des entiers.
  - on représente un argument par un entier:
    - texte: codage des lettres,
    - image: codage de la couleurs des pixels,
    - base de données: codage des champs, ...
  - les fonctions qu'on manipule opèrent sur des entiers (représentant des données argument) et renvoie un entier (représentant des données résultat).

Est-ce raisonnable ?

- du point de vue de la machine: peut-être,
- du point de vue du programmeur: non.

Données

- Informatique: science de l'information et du calcul.
- Données: information accessible, sous différentes formes:
  - nombres, textes, base de données, image, ...
- Cours 05: Cas particulier du texte.
- Comment exécuter des algorithmes sur des textes ?
  - chercher un motif, remplacer une lettre, compters des mots, ...
  - intuition: boucles itérées sur chaque lettre du texte.
- Informatique moderne: beaucoup de manipulations textuelles:
  - le web (réseaux sociaux, Wikipedia, ...),
  - la biologie (étude du génome),
  - la littérature, ...



Structures de Données

Les Cours 05 à 11 portent (presque) intégralement sur l'introduction de *structures de données*.

Une structure de données est une entité informatique qui regroupe et organise des données (ses éléments).

- l'interface est un ensemble de primitives du langage qui permettent son utilisation:
  - de construction: fabriquer une structure à partir de données.
  - de destruction: récupérer des données depuis structure.
  - d'utilisation: manipuler directement une ou plusieurs structures.
- l'implémentation est le code dans l'interpréteur (ou le compilateur) du langage qui implémente l'interface.
  - en LU1IN001, seule nous intéresse son efficacité
  - (Cours06: `=` + vs. `append`, Cours 09: itération vs. accès direct).
- on peut (souvent) imbriquer des structures de données.
  - i.e. considérer une structure dont les éléments sont eux-mêmes des structures.

Structures de Données: Exemple

Structure de données: les ensembles mathématiques d'entiers naturels:

- interface:
  - construction explicite: $\emptyset, \{1, 2, 3\}$
  - construction implicite: $\{(k, n),\ k \leq n \leq 100 \text{ tels que } k|n\}$
  - destruction par le minimum: $min(\{1, 2, 3\}) = 1$
  - utilisation, comparaison: $\{2, 2, 3\} = \{3, 2\}$
  - utilisation, union: $\{1, 2, 3\} \cup \{1, 4\} = \{1, 2, 3, 4\}$

Structures de Données: Exemple

Structure de données: les ensembles mathématiques d'entiers naturels:

- interface:
  - construction explicite: $\emptyset, \{1, 2, 3\}$
  - construction implicite: $\{(k, n),\ k \leq n \leq 100 \text{ tels que } k|n\}$
  - destruction par le minimum: $min(\{1, 2, 3\}) = 1$
  - utilisation, comparaison: $\{2, 2, 3\} = \{3, 2\}$
  - utilisation, union: $\{1, 2, 3\} \cup \{1, 4\} = \{1, 2, 3, 4\}$
- implémentation: les ensembles `set[int]` existent en *Python 101*.
  - interface au Cours 09,
  - quelques mots en compléments au Cours 09.
- imbrication: on peut considérer des ensembles d'ensembles $\{\emptyset,\ \{\emptyset\},\ \{\emptyset, \{\emptyset\}\},\ \ldots\}$

Structures de Données: Exemple

Structure de données: les ensembles mathématiques d'entiers naturels:

- interface:
  - construction explicite: $\emptyset, \{1, 2, 3\}$
  - construction implicite: $\{(k, n),\ k \leq n \leq 100 \text{ tels que } k|n\}$
  - destruction par le minimum: $min(\{1, 2, 3\}) = 1$
  - utilisation, comparaison: $\{2, 2, 3\} = \{3, 2\}$
  - utilisation, union: $\{1, 2, 3\} \cup \{1, 4\} = \{1, 2, 3, 4\}$
- implémentation: les ensembles `set[int]` existent en *Python 101*.
  - interface au Cours 09,
  - quelques mots en compléments au Cours 09.
- imbrication: on peut considérer des ensembles d'ensembles $\{\emptyset,\ \{\emptyset\},\ \{\emptyset, \{\emptyset\}\},\ \ldots\}$
  - ... mais pas en Python.

Structures de Données: Séquences

- Structures de données séquentielles (*aka* séquences).

  - une structure rassemble séquentiellement (i.e. à la suite) un nombre arbitraire fini de données similaires (de mêmes type).
  - organisation simple (ordre total) des données de la structure.
  - notion d'élément suivant: permet un parcours.
  - implémentation: facilité d'accès à l'élément suivant dans la mémoire.
- en LU1IN001 des séquences:
  - de nombres entiers consécutifs: intervalles de type `range`.
  - de caractères (lettres): chaînes de type `str`.
  - d'éléments d'un même type quelconque α: listes de type `list`$[\alpha]$ (C6)
- Utilisation particulière aux séquences:
  - itération: répéter une tâche pour chaque élément d'une séquence fixée, dans l'ordre de la séquence.

Intervalles

Définition

Un intervalle est une séquence d'entiers consécutifs, ordonnés par l'ordre standard.
Son type est `range`.

- au programme: intervalles d'entiers consécutifs croissants.
- manipulation:
  - construction directe d'un intervalle,
  - pas de destruction,
  - utilisation: itération sur un intervalle.

Syntaxe

l'expression `range(m, n)` construit l'intervalle des entiers relatifs compris entre `m` inclus à `n` exclus.

- `range(m, n)` est $[\![m; n[\![$ ou $[\![m; n-1]\!]$
- il y a $n - m$ éléments dans `range(m, n)`
- les intervalles sont des expressions (complexes) auto-évaluées.
  - `range(-2 - 3, 5 * 5)`
  - utilité ?

Intervalles

Définition

Un intervalle est une séquence d'entiers consécutifs, ordonnés par l'ordre standard.
Son type est `range`.

- au programme: intervalles d'entiers consécutifs croissants.
- manipulation:
  - construction directe d'un intervalle,
  - pas de destruction,
  - utilisation: itération sur un intervalle.

Syntaxe

l'expression `range(m, n)` construit l'intervalle des entiers relatifs compris entre `m` inclus à `n` exclus.

- `range(m, n)` est $[\![m; n[\![$ ou $[\![m; n-1]\!]$
- il y a $n - m$ éléments dans `range(m, n)`
- les intervalles sont des expressions (complexes) auto-évaluées.
  - `range(-2 - 3, 5 * 5)`
  - utilité ? itération

Itération sur une séquence

- Objectif: utiliser une séquence pour répéter une action pour chacun de ses éléments, dans l'ordre de la séquence.
- Objectif différent de celui de la boucle :
  - la boucle répète une action tant qu'une certaine condition est vraie.
  - l'itération sur une séquence répète une action pour tous les éléments d'une séquence.
- Terminaison ?

Itération sur une séquence

- Objectif: utiliser une séquence pour répéter une action pour chacun de ses éléments, dans l'ordre de la séquence.
- Objectif différent de celui de la boucle :
  - la boucle répète une action tant qu'une certaine condition est vraie.
  - l'itération sur une séquence répète une action pour tous les éléments d'une séquence.
- Terminaison ? toujours (séquence finie).

Syntaxe

La construction

```
for var in seq:
    corps
```

répète le `corps` (une suite d'instructions) de la boucle pour chaque élément dans la séquence `seq`, dans l'ordre.
L'élément `var` de `seq` peut apparaître (comme une occurence de variable) dans `corps` avec `var`.

Itération sur une séquence: principe d'interprétation

Pour interpréter:

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

1. On évalue `seq` en `s`.
2. Si `s` est vide on quitte l'itération et interprète `instr_s`
3. Sinon, on crée une nouvelle variable `var` de type `t` des éléments de `s`, initialisée au premier élément de `s`
4. On interprète les instructions `instr_k` du corps dans l'ordre.
   - elle peuvent lire la valeur de `var`.
5. Si il y a un élément suivant `var` dans `s`:
   - 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 un intervalle: principe d'interprétation

Pour interpréter:

```
i : int
for i in range(m, n):
    instr_1
    instr_2
    ...
    instr_n
instr_s
```

1. On évalue `range(m, n)` (en lui-même).
2. Si `m` $>$ `n`, on quitte l'itération et interprète `instr_s`
3. Sinon on crée une nouvelle variable entière `i` initialisée à `m`
4. On interprète les instructions `instr_k` du corps dans l'ordre.
   - elle peuvent utiliser `i`.
5. Si `i + 1` est inférieur strictement à `n`:
   - 5.1 on met `i + 1` dans `var`.
   - 5.2 on revient en 4.
6. Sinon:
   - on détruit `i`.
   - on sort de l'itération et on interprète `instr_s`

Somme des entiers par itération

Utilisation de **while**:

```
def somme_entier(n : int) -> int :
    """ Precondition : n >= 0
    renvoie la somme des n premiers entiers naturels."""

    i : int = 1
    s : int = 0

    while i <= n:
        s = s + i
        i = i + 1
    return s
```

Utilisation de **for**:

```
def somme_entier(n : int) -> int:
    """ Precondition : n >= 0
    renvoie la somme des n premiers entiers naturels."""

    s : int = 0

    i : int
    for i in range(1, n+1):
        s = s + i
    return s
```

- On itère sur l'intervalle $[\![1; n+1[\![$.
- Typage: on déclare la variable d'itération juste avant la boucle.

Simulation d'itération sur un intervalle

```python
def somme_entier(n : int) -> int:
    """ Precondition : n >= 0
    renvoie la somme des n premiers entiers naturels."""

    s : int = 0

    i : int
    for i in range(1, n+1):
        s = s + i
    return s
```

- Principe similaire à celui des boucles
  - on connaît la taille du tableau à l'avance,
  - la variable d'itération n'existe pas en dehors de la boucle.

```
somme_entiers(5)
```

Simulation d'itération sur un intervalle

```python
def somme_entier(n : int) -> int:
    """ Precondition : n >= 0
    renvoie la somme des n premiers entiers naturels."""

    s : int = 0

    i : int
    for i in range(1, n+1):
        s = s + i
    return s
```

- Principe similaire à celui des boucles
  - on connaît la taille du tableau à l'avance,
  - la variable d'itération n'existe pas en dehors de la boucle.

`somme_entiers(5)`

tour de boucle	variable i	variable s
entrée	-	0
1	1	1
2	2	3
3	3	6
4	4	10
5	5	15
sortie	-	15

Intervalles avancés

Raccourcis

`range(n)` s'évalue en `range(0,n)`.

Pas différent de 1

On peut spécifier des séquences ordonnées d'entiers, pas forcément consécutifs, en précisant leur pas:

- `range(5, 1, -1)`: intervalle décroissant.
- `range(0, 21, 2)`: intervalle d'entiers pairs.

- Attention aux pas non-entiers.
  - `range(0, 1, 0.1)` n'existe pas.
- en LU1IN001 n'utiliser que `range(n,m)` et `range(n)`

Encodage des intervalles

Les intervalles entiers ne rajoutent pas d'expressivité à notre langage.

```
i : int
for i in range(m, n):
    instr_1
    instr_2
    ...
    instr_n
instr_s
```

$\Leftrightarrow$

```
i : int = m

while i < n:
    instr_1
    instr_2
    ...
    instr_n
    i = i + 1
instr_s
```

- Correspondance entre les deux constructions.
- Elégance, concision (donc lisibilité) du code:

Encodage des intervalles

Les intervalles entiers ne rajoutent pas d'expressivité à notre langage.

```
i : int
for i in range(m, n):
    instr_1
    instr_2
    ...
    instr_n
instr_s
```

$\Leftrightarrow$

```
i : int = m

while i < n:
    instr_1
    instr_2
    ...
    instr_n
    i = i + 1
instr_s
```

- Correspondance entre les deux constructions.
- Elégance, concision (donc lisibilité) du code: avantage à l'itération.
- Légère différence:

Encodage des intervalles

Les intervalles entiers ne rajoutent pas d'expressivité à notre langage.

```
i : int
for i in range(m, n):
    instr_1
    instr_2
    ...
    instr_n
instr_s
```

$\Leftrightarrow$

```
i : int = m

while i < n:
    instr_1
    instr_2
    ...
    instr_n
    i = i + 1
instr_s
```

- Correspondance entre les deux constructions.
- Elégance, concision (donc lisibilité) du code: avantage à l'itération.
- Légère différence: occurence de `i` dans `instr_s` ?

```
j : int = 0
...
i : int
for i in range(m, n):
    instr_1
    ...
    instr_n
    j = i
instr_s
```

Encodage des caractères

- Ecriture: arrangement séquentiel de symboles, issu d'un alphabet; composant un texte.
  - un symbole = un son, avec (romain) ou sans voyelle (arabe),
  - un symbole = une syllabe (hiragana),
  - un symbole = un mot (kanji).
- Codage des symboles de l'alphabet:
  - à la main: ensemble de traits (romain), ensemble de traits dirigés (kanji), matrice de points (braille), ...
  - imprimerie: notion de caractères (atomes d'impression).
- ASCII: codage de certains symboles (romain, ROMAIN, chiffres, ...) sur 7 bits (nombres de 0 à 127).
  - `a` $\longrightarrow$ `97` $\longrightarrow$ `1100001`
  - `R` $\longrightarrow$ `82` $\longrightarrow$ `1010010`
  - `%` $\longrightarrow$ `37` $\longrightarrow$ `0100101`
- Unicode: codage sur un nombre de bytes (8 bits) flottant:
  - tres inclusif
    - beaucoup d'alphabet connus (tifinagh, malayalam, ...)
    - propositions farfelues (tengwar, klingon, ...)
  - compatible avec ASCII.
  - en Python 101:
    - `chr` donne le caractère associé à un entier en Unicode.
    - `ord` donne l'entier associé à un caractère par Unicode.

Chaînes de caractères

Définition

Une chaîne de caractères est une séquence de caractères.
Son type est `str`.
Un caractère est un symbole atomique reconnu par l'Unicode:

- Représenter des textes
  - dans toutes les langues
- Manipulation de chaînes de caractères:
  - construire,
  - détruire (récupérer un caractère, une sous-chaîne),
  - utiliser: itérer, comparer.
- Ubiquitaires dans l'informatique moderne.

Construction

Deux méthodes pour construire des chaînes:

- construction explicite: expression atomique de chaîne,
- construction inductive: concaténation.

Expressions atomiques de chaîne

Ecriture des symboles de la chaînes, dans l'ordre, entre guillemets:

- `'Longtemps, je me suis couche de bonne heure.'`
- `"C'etait a Megara, faubourg de Carthage, dans les jardins d'Hamilcar."`
  - utilisation du caractère `'` dans la chaîne.
  - comparer avec

    `'C'etait a Megara, faubourg de Carthage, dans les jardins d'Hamilcar.'`
- `'J\'avais vingt ans. Je ne laisserai jamais personne dire ...'`
  - Echappement de `'` avec un \. (x)

Remarques:

- un caractère est aussi un objet de type `str` (pas comme en C)
- ne pas confondre chaînes et autres types:
  - `return "True"`

Concaténation

Construction par concaténation

L'opérateur primitif `+` qui prend deux `str` et renvoie une `str` concatène les chaînes de caractères.

- `"All this happened, "+ "more or less."` donne
  `"All this happened, more or less."`
- associativité: `'a'+ ('b'+ 'c')` et `('a'+ 'b') + 'c'`
- non-commutativité: `'bonnet'+ 'blanc'` et `'blanc'+ 'bonnet'`
- neutralité de `''` (chaine vide): `'' + 'saucisse'`, `'saucisse'+ ''`, `'saucisse'`
- surcharge de `+`: `'2' + '3'` et `2 + 3`

Répétition

Problème

Donner une fonction `repetition` qui concatène bout à bout plusieurs fois la même chaîne de caractère.

- `repetition('po', 4)` donne `'popopopo'`
- `repetition('Boutros ', 2) + 'Ghali'` donne `'Boutros Boutros Ghali'`

Répétition

Problème

Donner une fonction `repetition` qui concatène bout à bout plusieurs fois la même chaîne de caractère.

- `repetition('po', 4)` donne `'popopopo'`
- `repetition('Boutros ', 2) + 'Ghali'` donne `'Boutros Boutros Ghali'`

```python
def repetition(s : str, n : int) -> str:
    """ Precondtion : n >= 1
    retourne la chaine composee de n repetitions
    successives de la chaine s."""

    r : str = ''  # on initialise avec la chaine vide puisque c'est l'element neutre

    i : int # (caractere courant)
    for i in range(1, n + 1):
        r = r + s

    return r
```

Répétition (Simulation)

Simulation de `repetition('bla',3)`

Répétition (Simulation)

Simulation de `repetition('bla',3)`

tour de boucle	variable i	variable r
entrée	-	`''`
1	1	`'bla'`
2	2	`'blabla'`
3	3	`'blablabla'`
sortie	-	`'blablabla'`

Répétition (Simulation)

Simulation de `repetition('bla',3)`

tour de boucle	variable i	variable r
entrée	-	`''`
1	1	`'bla'`
2	2	`'blabla'`
3	3	`'blablabla'`
sortie	-	`'blablabla'`

- Répétition est définie en Python 101 par la primitive `*`
  - on peut écrire `return s * n`
  - efficacité ?
- On peut évidemment utiliser `while` pour `repetition`.

Destruction

Récupérer des éléments d'une chaîne:

- des caractères,
- des sous-chaînes.

Destruction en caractères

On peut récupérer un caractère d'une chaîne par son indice.
Le (i+1)ème caractère de la chaîne `s` (en lisant de gauche à droite) se récupère avec `s[i]`.

Exemple, chaîne `s = 'Yes we can'`

Caractères	`'Y'`	`'e'`	`'s'`	`' '`	`'w'`	`'e'`	`' '`	`'c'`	`'a'`	`'n'`
indice	0	1	2	3	4	5	6	7	8	9

- `s[0]` donne `'Y'`,
- `s[9]` donne `'n'`,
- `s[10]` produit une erreur.

Destruction (II)

Destruction en caractères, indice inverse

Le ième caractère de la chaîne `s` (en lisant de droite à gauche) se récupère avec `s[-i]`.

Exemple, chaîne `s = 'Yes we can'`

Caractère	`'Y'`	`'e'`	`'s'`	`' '`	`'w'`	`'e'`	`' '`	`'c'`	`'a'`	`'n'`
indice	0	1	2	3	4	5	6	7	8	9
indice inverse	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

- `s[-1]` donne `'n'`,
- `s[-9]` donne `'e'`,
- `s[-12]` lève une erreur.

Découpage

Le but du découpage (*slicing*) est d'obtenir une sous-chaîne d'une chaîne donnée.

Syntaxe

Le découpage d'une chaîne `s` entre le (i+1)ème (inclus) et le (j+1)ème (exclus) (de gauche à droite) s'obtient par `s[i : j]`

- correspond au découpage entre les indices `i` (inclus) et `j` (exclus)
- même inclusion/exclusion de bornes que dans les intervalles.
- `s[i:i+1]` donne le même résultat que `s[i]`

Exemple, chaîne `s = 'Yes we can'`

Caractère	`'Y'`	`'e'`	`'s'`	`' '`	`'w'`	`'e'`	`' '`	`'c'`	`'a'`	`'n'`
indice	0	1	2	3	4	5	6	7	8	9
indice inverse	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

- `s[0:3]` donne `'Yes'`,
- `s[4:6]` donne `'we'`,
- `s[-10:-7]` donne `'Yes'`.
- `s[5:3]` donne `''`.

Utilisation: Comparaison

Comparaisons de chaînes

Les chaînes de caractère peuvent être comparées:

- égalité: `== : str * str -> bool`
- inégalité: `!= : str * str -> bool`

- Principe d'évaluation de l'égalité:
  1. Si les chaînes ont des longueurs différentes on évalue en `false`.
  2. Sinon, on commence au premier caractères des deux chaînes:
     - 2.1 Si il est différent, on évalue en `false`,
     - 2.2 Sinon on passe au caractère suivant dans les deux chaînes.
  3. Quand on a parcouru les deux chaînes en entier on évalue en `true`.
- Indifférence aux guillemets `'saucisse'` et `"saucisse"`
- Attention à la casse `'saucisse'` et `'Saucisse'`

Complément: Ordre sur les chaînes

Les opérateurs `<`, `>`, `<=`, `>=` sont aussi surchargés en `str * str -> str`.

Comparaisons de chaînes

Soient `s1` et `s2` deux chaînes de caractère:

- `s1 < s2` quand `s1` est strictement plus petite que `s2` dans l'ordre lexicographique.

- Ordre lexicographique:
  - ordre du dictionnaire,
  - comparaison itérative des lettres des deux mots:
    - on part de la première lettre,
    - si identique, on passe aux suivantes,
    - et ainsi de suite ...
- `'azitrhomycine' < 'baba'`
- l'absence de caractères précède dans l'alphabet: `'bon' < 'bonbon'`
- comparaison de caractères:
  - ordre des numéros Unicode,
  - majuscules d'abord: `'Z' < 'a'`,
  - symboles non-alphabétiques: `'#' < '{'`

Réduction

La réduction est un type de problème.

Définition

La réduction d'une séquence, consiste à construire une information structurellement plus simple en parcourant la séquence.

- pour les chaînes, produire un caractère, ou `int` ou `bool`
  - extraire un caractère d'une chaîne,
  - compter quelque chose dans une chaîne,
  - décider si une chaîne valide une propriété.
- deux approches pour la solution itérative (parcours):
  - par itération des éléments de la chaîne (chaîne = séquence),
  - par itération des indices de la chaîne (indices = intervalle).

Itération sur une chaîne: principe d'interprétation

Pour interpréter:

```
for c in s:
    instr_1
    instr_2
    ...
    instr_n
instr_s
```

1. On évalue `s`
2. Si `s` est `''` on quitte l'itération et interprète `instr_s`
3. Sinon, on crée une nouvelle variable `c` initialisée au caractère d'incide `0` de `s`
4. On interprète les instructions `instr_i` du corps dans l'ordre.
   - elle peuvent faire apparaître `c`.
5. Si il y a un caractère suivant dans `seq`:
   5.1 on met le caractère suivant dans `c`.
   5.2 on revient en 4.
6. Sinon:
   - on détruit `c`.
   - on sort de l'itération et on interprète `instr_s`

Longueur d'une chaîne

Problème

Donner une fonction `longueur` qui prend en entrée une chaîne `s` et renvoie la longueur de `s`, c'est à dire le nombre de ses caractères.

- son en-tête est est `longueur(s : str) -> int`
- `longueur('Heureux qui, comme Ulysse, a fait un beau voyage,')` donne `49`
- `longueur('49')` donne `2`
- algorithme:

Longueur d'une chaîne

Problème

Donner une fonction `longueur` qui prend en entrée une chaîne `s` et renvoie la longueur de `s`, c'est à dire le nombre de ses caractères.

- son en-tête est est `longueur(s : str) -> int`
- `longueur('Heureux qui, comme Ulysse, a fait un beau voyage,')` donne `49`
- `longueur('49')` donne `2`
- algorithme: on veut compter chaque caractère.
  - on parcourt les caractères de la chaîne,
  - on ajoute 1 dans une variable à chaque caractère.

Longueur d'une chaîne

Problème

Donner une fonction `longueur` qui prend en entrée une chaîne `s` et renvoie la longueur de `s`, c'est à dire le nombre de ses caractères.

- son en-tête est est `longueur(s : str) -> int`
- `longueur('Heureux qui, comme Ulysse, a fait un beau voyage,')` donne `49`
- `longueur('49')` donne `2`
- algorithme: on veut compter chaque caractère.
  - on parcourt les caractères de la chaîne,
  - on ajoute 1 dans une variable à chaque caractère.

```
def longueur(s : str) -> int:
    """retourne la longueur de la chaine s."""

    l : int = 0

    c : str
    for c in s:
        l = l + 1

    return l
```

(existe en Python 101: `len`)

Nombre d'occurences d'un caractère

Problème

Donner une fonction `occurences` qui prend en entrée un caractère `c` et une chaîne `s` et renvoie le nombre d'occurence de `c` dans `s`.

- son en-tête est `occurences(c : str, s : str) -> int`
- `occurences('e','Heureux qui, comme Ulysse, a fait un beau voyage,')` donne `6`
- algorithme:

Nombre d'occurences d'un caractère

Problème

Donner une fonction `occurences` qui prend en entrée un caractère `c` et une chaîne `s` et renvoie le nombre d'occurence de `c` dans `s`.

- son en-tête est `occurences(c : str, s : str) -> int`
- `occurences('e','Heureux qui, comme Ulysse, a fait un beau voyage,')` donne `6`
- algorithme: on veut compter un caractère quand il vaut `c`.
  - on parcourt les caractères de la chaîne,
  - on ajoute 1 dans une variable à chaque caractère qui vaut `c`.

Nombre d'occurences d'un caractère

Problème

Donner une fonction `occurences` qui prend en entrée un caractère `c` et une chaîne `s` et renvoie le nombre d'occurence de `c` dans `s`.

- son en-tête est `occurences(c : str, s : str) -> int`
- `occurences('e','Heureux qui, comme Ulysse, a fait un beau voyage,')` donne `6`
- algorithme: on veut compter un caractère quand il vaut `c`.
  - on parcourt les caractères de la chaîne,
  - on ajoute 1 dans une variable à chaque caractère qui vaut `c`.

```
def occurrences(c : str, s : str) -> int:
    """Precondition : len(c) == 1
    retourne le nombre d'occurrences du caractere c dans la chaine s"""

    nb : int = 0

    d : str
    for d in s:
        if d == c:
            nb = nb + 1

    return nb
```

- Précondition pour s'assurer que `c` est un caractère.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Itération sur l'intervalle des indices

- `range(len(s))` donne l'intervalle des indices de la chaîne `s`
- on peut accéder au caractère d'indice courant `i` de `s` par `s[i]`

```
def occurences(c : str, s : str) -> int:
    """Precondition : len(c) == 1 """

    nb : int = 0

    i : int
    for i in range(len(s)):
        if s[i] == c:
            nb = nb + 1
    return nb
```

Présence d'un caractère

Problème

Donner une fonction `presence` qui prend en entrée un caractère `c` et une chaîne `s` et renvoie `true` si `c` appartient à `s` et `false` sinon.

- algorithme:

Présence d'un caractère

Problème

Donner une fonction `presence` qui prend en entrée un caractère `c` et une chaîne `s` et renvoie `true` si `c` appartient à `s` et `false` sinon.

- algorithme: un caractère est présent quand `occurences` est ≥ 1.

Présence d'un caractère

Problème

Donner une fonction `presence` qui prend en entrée un caractère `c` et une chaîne `s` et renvoie `true` si `c` appartient à `s` et `false` sinon.

- algorithme: un caractère est présent quand `occurences` est ≥ 1.

```
def presence(c : str, s : str) -> bool:
    """Precondition : len(c) == 1
    retourne True si le caractere c est present dans la chaine s,
    ou False sinon"""

    return occurrences(c, s) > 0
```

- Efficacité ?
  - `presence('e', 'Heureux qui, comme Ulysse, a fait un beau voyage,')`

Présence d'un caractère

Problème

Donner une fonction `presence` qui prend en entrée un caractère `c` et une chaîne `s` et renvoie `true` si `c` appartient à `s` et `false` sinon.

- algorithme: un caractère est présent quand `occurences` est ≥ 1.

```
def presence(c : str, s : str) -> bool:
    """Precondition : len(c) == 1
    retourne True si le caractere c est present dans la chaine s,
    ou False sinon"""

    return occurrences(c, s) > 0
```

- Efficacité ?
  - `presence('e', 'Heureux qui, comme Ulysse, a fait un beau voyage,')`
- Solution: sortie anticipée.

Présence d'un caractère (II)

```python
def presence(c : str, s : str) -> bool:
    """Precondition : len(c) == 1
    Retourne True si le caractere c est present dans la chaine s,
    ou False sinon"""

    d : str
    for d in s:
        if d == c:
            return True

    return False
```

```
presence('e', 'Heureux qui, comme Ulysse, a fait un beau voyage,')
```

Présence d'un caractère (II)

```python
def presence(c : str, s : str) -> bool:
    """Precondition : len(c) == 1
    Retourne True si le caractere c est present dans la chaine s,
    ou False sinon"""

    d : str
    for d in s:
        if d == c:
            return True

    return False
```

`presence('e', 'Heureux qui, comme Ulysse, a fait un beau voyage,')`

tour de boucle	variable d
entrée	-
1	'H'
2	'e'
sortie anticipée	-

Recherche par parcours des indices

Problème

Donner une fonction `premier_indice` qui prend en entrée un caractère `c` et une chaîne `s` et qui renvoie le premier indice d'occurence de `c` dans `s` s'il existe (et rien sinon).

- fonction partielle `premier_indice(c : str, s: str) -> Optional[int]`
  - `Optional[int]` signifie "soit `int`, soit `None`"
- `premier_indice('u', 'Heureux qui, comme Ulysse, a fait un beau voyage,')` donne `2`
- `premier_indice('z', 'Heureux qui, comme Ulysse, a fait un beau voyage,')` donne `None`
- on veut se souvenir de l'indice dans un tour de boucle.
  - impossible d'utiliser l'itération sur les caractères.

Recherche avec boucle

```python
def premier_indice(c : str, s :str) -> Optional[int]:
    """Precondition: len(c) == 1"""

    i : int
    for i in range(len(s)):
        if s[i] == c:
            return i
```

```python
def premier_indice(c : str, s : str) -> Optional[int]:
    """Precondition : len(c) == 1"""

    i : int = 0
    trouve : bool = False

    while (not trouve) and i < len(s):
        if s[i] == c:
            trouve = True
        i = i + 1
    if trouve:
        return i-1
```

- Sortie anticipée,
- `return None` à la fin est facultatif.

Transformation et Filtrage

D'autres types de problèmes.

Définition

La transformation ou le filtrage d'une séquence consiste à construire une autre séquence en parcourant la première.

- type habituel: `str` –> `str`
- transformation: la sortie est une autre chaîne de même longueur (action à chaque caractère),
- filtrage: la sortie est une sous-chaîne (pas forcément d'éléments consécutifs, mais dans le même ordre).
- la chaîne en paramètre n'est pas modifiée.
  - on ne modifie pas les paramètres en Python 101,
  - en Python, `str` est immutable.
- on construit itérativement la chaîne résultat.

Substitution

Problème

Donner une fonction substitution qui prend en entrée deux caractères `c` et `d` et une chaîne `s` et qui renvoie une chaîne corresondant à `s` dans laquelle on a remplacé toutes les occurences de `c` par `d`.

- Attention: `s` n'est pas modifiée (fonctionnalité pure).
- en-tête: `substitution(c : str, d : str, s : str) -> str`
- `substitution('e', 'i', 'Heureux qui, comme Ulysse, a fait un beau voyage,')` donne `'Hiuriux qui, commi Ulyssi, a fait un biau voyagi,'`

Substitution

Problème

Donner une fonction substitution qui prend en entrée deux caractères `c` et `d` et une chaîne `s` et qui renvoie une chaîne corresondant à `s` dans laquelle on a remplacé toutes les occurences de `c` par `d`.

- Attention: `s` n'est pas modifiée (fonctionnalité pure).
- en-tête: `substitution(c : str, d : str, s : str) -> str`
- `substitution('e', 'i', 'Heureux qui, comme Ulysse, a fait un beau voyage,')` donne `'Hiuriux qui, commi Ulyssi, a fait un biau voyagi,'`

```
def substitution(c : str, d : str, s : str) -> str:
    """Precondition: (len(c) == 1) and (len(d) == 1)
    renvoie la chaine resultant de la substitution du caractere c par
    le caractere d dans la chaine s."""

    r : str = ''

    e : str
    for e in s:
        if e == c:
            r = r + d  # on substitue
        else:
            r = r + e # on garde e

    return r
```

Suppression

Problème

Donner une fonction `suppression` qui prend en entrée un caractère `c` et une chaîne `s` et qui renvoie une chaîne correspondant à `s` dans laquelle on a retiré toutes les occurences de `c`.

- filtre: prédicat à valider pour être dans la chaîne résultat
  - ici, être différent de `c`
- `suppression('e', 'Heureux qui, comme Ulysse, a fait un beau voyage,')` donne `'Hurux qui, comm Ulyss, a fait un bau voyag,'`

Suppression

Problème

Donner une fonction `suppression` qui prend en entrée un caractère `c` et une chaîne `s` et qui renvoie une chaîne correspondant à `s` dans laquelle on a retiré toutes les occurences de `c`.

- filtre: prédicat à valider pour être dans la chaîne résultat
  - ici, être différent de `c`
- `suppression('e', 'Heureux qui, comme Ulysse, a fait un beau voyage,')` donne `'Hurux qui, comm Ulyss, a fait un bau voyag,'`

```python
def suppression(c : str, s : str) -> str:
    """Precondition: len(c) == 1
    renvoie la sous-chaine de s dans laquelle toutes
    les occurrences du caractere c ont ete supprimees."""

    r : str = ''

    d : str
    for d in s:
        if d != c:
            r = r + d  # ne pas supprimer
        # sinon ne rien faire (supprimer)

    return r
```

Inversion

Problème

Donner une fonction `inversion` qui prend en entrée une chaîne `s` et renvoie une chaîne correspondant à l'inversion de `s` (lecture de droite à gauche).

- `inversion('Heureux qui, comme Ulysse, a fait un beau voyage,')` donne `',egayov uaeb nu tiaf a ,essylU emmoc ,iuq xuerueH'`

Inversion

Problème

Donner une fonction `inversion` qui prend en entrée une chaîne `s` et renvoie une chaîne correspondant à l'inversion de `s` (lecture de droite à gauche).

- `inversion('Heureux qui, comme Ulysse, a fait un beau voyage,')` donne `',egayov uaeb nu tiaf a ,essylU emmoc ,iuq xuerueH'`

```
def inversion(s : str) -> str:
    """retourne la chaine s."""

    r : str = ''


    c : str
    for c in s:
        r = c + r # le caractere courant c est place
                  # au debut de la nouvelle chaine en construction

    return r
```

Inversion (II)

- on parcourt `s` et on ajoute le caractère courant en tête de la chaîne résultat.

Simulation de `inversion('saucisse')`

Inversion (II)

- on parcourt `s` et on ajoute le caractère courant en tête de la chaîne résultat.

Simulation de `inversion('saucisse')`

tour de boucle	variable c	variable r
entrée	-	`''`
1	`'s'`	`'s'`
2	`'a'`	`'as'`
3	`'u'`	`'uas'`
4	`'c'`	`'cuas'`
5	`'i'`	`'icuas'`
6	`'s'`	`'sicuas'`
7	`'s'`	`'ssicuas'`
8	`'e'`	`'essicuas'`
sortie	-	`'essicuas'`

Entrelacement

Problème

Donner une fonction `entrelacement` qui prend en entrée une chaîne `s1` et une chaîne `s2` et qui renvoie une chaîne correspondant à entrelacement de `s1` et `s2`.

- `entrelacement('saucisse', 'chocolat')` donne `'scahuoccioslsaet'`
- `entrelacement('saucisse', 'aaa')` donne `'saaauacisse'`

Entrelacement

Problème

Donner une fonction `entrelacement` qui prend en entrée une chaîne `s1` et une chaîne `s2` et qui renvoie une chaîne correspondant à entrelacement de `s1` et `s2`.

- `entrelacement('saucisse', 'chocolat')` donne `'scahuoccioslsaet'`
- `entrelacement('saucisse', 'aaa')` donne `'saaauacisse'`

```
def entrelacement(s1 : str, s2 : str) -> str:
    """renvoie la chaine constituee par l'entrelacement
    des caracteres des chaines s1 et s2."""

    i : int = 0  # indice pour parcourir les deux chaines

    r : str = ''  # chaine resultat

    while (i < len(s1)) and (i < len(s2)):
        r = r + s1[i] + s2[i]
        i = i + 1

    if i < len(s1):
        r = r + s1[i:len(s1)]
    elif i < len(s2):
        r = r + s2[i:len(s2)]

    return r
```

Entrelacement (II)

- tant que l'indice est plus petit que la longueur des deux chaînes, on ajoute les caractères courants.
- on complète à la fin.

Simulation de `entrelacement('saucisse', 'cafe')`

Entrelacement (II)

- tant que l'indice est plus petit que la longueur des deux chaînes, on ajoute les caractères courants.
- on complète à la fin.

Simulation de `entrelacement('saucisse', 'cafe')`

tour de boucle	**variable i**	`s1[i]`	`s2[i]`	`variable r`
entrée	0	`'s'`	`'c'`	`''`
1	1	`'s'`	`'c'`	`'sc'`
2	2	`'a'`	`'a'`	`'scaa'`
3	3	`'u'`	`'f'`	`'scaauf'`
4 (sortie)	4	`'c'`	`'e'`	`'scaaufce'`

- en sortie fonction on concatene `'isse'` pour obtenir `'scaaufceisse'`

Itération vs. Boucle

- Toute itération peut-être représentée par une boucle:

```
def suppr(c : str, s : str) -> str:
    """Precondition: len(c) == 1"""

    r : str = ''

    d : str
    for d in s:
        if d != c:
            r = r + d

    return r
```

⇒

Itération par éléments.

```
def suppr(c : str, s :str) -> str:
    """Precondition: len(c) == 1"""

    r : str = ''

    i : int
    for i in range(0, len(s)):
        if s[i] != c:
            r = r + s[i]

    return r
```

⇒

Itération par indices.

```
def suppr(c : str, s : str) -> str:
    """Precondition: len(c) == 1"""

    r : str = ''
    i : int = 0

    while i < len(s):
        if s[i] != c:
            r = r + s[i]
        i = i + 1

    return r
```

Boucle.

Itération vs. Boucle

- Toute itération peut-être représentée par une boucle:

```
def suppr(c : str, s : str) -> str:
    """Precondition: len(c) == 1"""

    r : str = ''

    d : str
    for d in s:
        if d != c:
            r = r + d

    return r
```

⇒

Itération par éléments.

```
def suppr(c : str, s :str) -> str:
    """Precondition: len(c) == 1"""

    r : str = ''

    i : int
    for i in range(0, len(s)):
        if s[i] != c:
            r = r + s[i]

    return r
```

⇒

Itération par indices.

```
def suppr(c : str, s : str) -> str:
    """Precondition: len(c) == 1"""

    r : str = ''
    i : int = 0

    while i < len(s):
        if s[i] != c:
            r = r + s[i]
        i = i + 1

    return r
```

Boucle.

- L'inverse n'est pas vraie.

```
def pgcd(a,b):
    """ ... """
    q = a
    r = b
    temp = 0
    while r != 0:
        temp = q % r
        q = r
        r = temp
    return q
```

- Comment trouver une itération correspondante ?
- Différence: borne fixe (itération) vs. condition d'arrêt (boucle).

Itération vs. Boucle (II)

- On préfère une itération quand c'est possible (élégance).
- L'efficacité est similaire.

```python
def factorielle(n : int) -> int:
    """Precondition : n > 0"""

    k : int = 1   # on demarre au rang 1
    f : int = 1   # factorielle au rang 1

    while k <= n:
        f = f * k
        k = k + 1

    return f
```

```python
def factorielle(n : int) -> int:
    """Precondition : n > 0"""

    f : int = 1   # on demarre au rang 1

    k : int
    for k in range(1, n + 1):
        f = f * k

    return f
```

Conclusion

- correction (invariant)
- terminaison (variant)
- efficacité
- séquences : intervalles
- séquences : chaînes de caractères
- Culture Générale :
  - Thèse de Church

Conclusion (II)

TD-TME 04

- Thèmes 05 du cahier d'exercices.

Activité 04

- Cryptographie

Cours 05 - 11/10/2021

- "ces listes qui nous gouvernent"

Conclusion

- Intervalles:
  - utilisation de `range(m, n)`
  - itération sur un intervalle d'entiers.
- Chaînes:
  - syntaxe,
  - opérations de construction,
  - opérations de destruction,
  - itération sur les chaînes,
  - résoudre des problèmes:
    - réduction,
    - transformation,
    - filtrage,
    - les autres.
- Ne pas oublier les boucles.

Cours 06 (10/10)

Ces Listes qui nous gouvernent.