Éléments de Programmation +
Cours 2 - Variables, Alternatives, Boucles

Romain Demangeon

LU1IN011 - Section ScFo 11 + 13 + ...

19/09/2022

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Distribution des polys

- La distribution prévue cette semaine est annulée.
- La distribution aura lieu a une date ultérieure inconnue.

Etat du cours

- On dispose d'un interpréteur du langage informatique *Python 101* : *MrPython*.
- L'interpréteur lit, analyse, comprend et apprend du code.
- La zone d'évaluation nous permet d'évaluer des expressions.
  - expressions atomiques / composées,
  - expressions arithmétiques,
  - utilisation de primitives (`math.sqrt`)
- La sone d'édition permet d'écrire ses propres fonctions.

Définition de Fonctions

- Avec seulement des primitives: calculatrice améliorée.
- Principe de la programmation: définition de fonctions par le programmeur (on "généralise" des calculs).
- Les fonctions ont une place centrale en informatique.
- Elles permettent de paramètrer, d'automatiser et de généraliser des calculs.

Définition de Fonctions: exemple

Problème: Calculer le périmètre d'un rectangle défini par sa longueur et sa largeur.

- formule mathématique: $2 * (la + lon)$
- si $la = 2$ et $lon = 3$, on saisit `2 * (2 + 3)`
- en mathématiques, on définirait la fonction périmètre par:

$$p : \begin{array}{ccc} \mathbb{N} \times \mathbb{N} & \rightarrow & \mathbb{N} \\ (la, lon) & \mapsto & 2 * (la + lon) \end{array}$$

-

```python
def perimetre(largeur : int, longueur : int) -> int:
    """Precondition : (longueur >= 0) and (largeur >= 0)
    Precondition : longueur >= largeur
    retourne le perimetre du rectangle defini par sa largeur et sa longueur."""

    return 2 * (largeur + longueur)
```

- on peut la tester avec `assert perimetre(2, 3) == 10`

Définition de fonction: étapes

Pour définir une fonction, on passe par les étapes suivantes:

1. Spécification du calcul effectué par la fonction:
   1.1 en-tête de la fonction, avec types.
   1.2 précondition pour son application.
2. Implémentation de l'algorithme calculant le résultat.
3. Validation de la fonction par un jeu de test.

```python
def perimetre(largeur : int, longueur : int) -> int:
    """Precondition : (longueur >= 0) and (largeur >= 0)
    Precondition : longueur >= largeur
    retourne le perimetre du rectangle defini par sa largeur et sa longueur."""

    return 2 * (largeur + longueur)

# Jeu de tests
assert perimetre(2, 3) == 10
assert perimetre(4, 9) == 26
assert perimetre(0, 0) == 0
assert perimetre(0, 8) == 16
```

Spécification

Définition

La spécification d'une fonction est la partie du code qui:

1. décrit le problème que la fonction résout.
2. décrit comment on utilise la fonction.

- rôle: permettre à un programmeur (y compris soi-même) de comprendre comment utiliser la fonction. (fondamental dans l'industrie)
- en-tête:
  - introduit par `def`
  - donne le nom de la fonction et ceux de ses paramètres.
  - donne le type des paramètres.
  - donne le type du résultat que calcule la fonction.
- indentation: 4 espaces/une tabulation.
- des préconditions
  - expressions logiques que doivent vérifier les paramètres.
  - le programmeur garantit le bon fonctionnement seulement quand les préconditions sont satisfaites.
- une description du calcul effectué par la fonction.

Implémentation

Définition

L'implémentation d'une fonction est l'écriture de l'algorithme qui calcule le résultat de la fonction dans un langage informatique.

- le corps de la fonction est composé d'instructions.
- premier cours: une unique instruction `return expr`
- évaluation de l'instruction `return expr`:
  1. On calcule la valeur de `expr`.
  2. On retourne à l'appelant de la fonction le résultat.
- Plusieurs solutions au problème que la fonction résout: d'autres implémentations.

```python
def perimetre(largeur : int, longueur : int) -> int:
    """Precondition : (longueur >= 0) and (largeur >= 0)
    Precondition : longueur >= largeur

    retourne le perimetre du rectangle defini par sa largeur et sa longueur."""

    return (largeur + longueur) + (largeur + longueur)
```

Validation

Approche Contractuelle

- un client avec un problème,
- un programmeur avec une solution.
  - solution = une fonction.

- Problème posé $\leftrightarrow$ Fonction pour le résoudre
- Solution: Définition (Spécification + Implémentation) + Validation
- Validation: montrer que la fonction "marche": calcule bien une solution au problème.
  - problème avec contrat,
  - solution avec tests.
- Appeler la fonction sur des arguments.
- Tester avec des arguments qui respectent la spécification.
- Jeu de tests:
  - expressions d'appel valides.
  - couvrir suffisamment de cas (difficile).

Etat du cours

- Expressions:
  - atomiques / composées,
  - comprises par l'interpréteur,
  - évaluées par l'interpréteur,
    - sémantique: régles d'évaluation.
- Fonctions:
  - une seule instruction: **return**
  - expressions paramétrées,
  - évaluation de l'appel de fonction,
- Expressivité:
  - uniquement des expressions qui se réduisent.
  - calculatrice d'expressions mathématiques,
  - expressivité suffisante avec la récursion:
    - programmation récursive/fonctionnelle

```python
def fact(n : int) -> int:
return 1 if (n == 0) else n * fact(n-1)
```

    - pas au programme.

Instructions

Définition

Une instruction est un ordre de calcul donné à la machine.
Le corps d'une fonction est une séquence d'instructions.

Instructions vs. Expressions

- une expression s'évalue en sa valeur.
- une instruction s'interprète (et n'a pas de valeur).
- une instruction contient (souvent) une (des) expression(s).
- évaluer une application, c'est interpréter le corps de la fonction.

Instruction `return`

Principe d'interprétation de `return expr`:

1. On évalue `expr` en sa valeur `v`.
2. On sort de la fonction avec comme valeur de retour `v`.

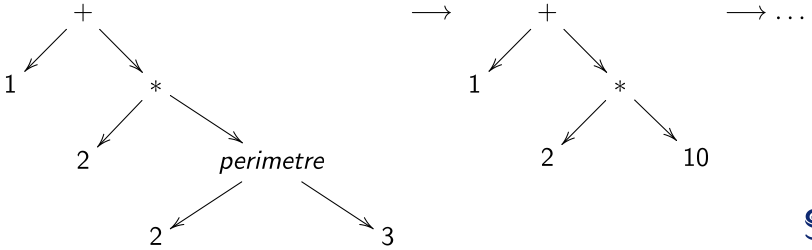
Valeur de retour

- La valeur de retour est le résultat de la fonction.
- La valeur de retour est envoyée à l'appelant.
- Si l'appelant est le *top-level* (fenêtre d'évaluation), il y a affichage.

```
=== Evaluation de : 'perimetre(2,3)' ===
10
========================================
```

- Si l'appelant est une expression, l'évaluation continue en remplaçant l'appel par la valeur de retour.

```
=== Evaluation de : '1 + 2 * perimetre(2, 3)' ===
21
=================================================
```



Appel d'une fonction dans une fonction

Problème: Calculer le périmètre d'un rectangle obtenu en accolant `nb` rectangles identiques par la largeur.

Solution:

Appel d'une fonction dans une fonction

Problème: Calculer le périmètre d'un rectangle obtenu en accolant `nb` rectangles identiques par la largeur.
Solution:

```
def perimetre_n(larg : float, long : float, nb : int) -> float:
    """ precondition: larg >= 0 and long >= 0
    precondition: nb > 0"""

    return nb * perimetre(larg, long) - (2 * nb - 2) * larg
```

- Comprendre l'évaluation de `perimetre_n(2, 3, 4)`:
  1. (Expr.) Evaluation de l'expression `perimetre_n(2, 3, 4)`.
  2. (Appel) Calcul de `perimetre_n`, `larg` vaut `2`, `long` vaut `3` et `nb` vaut `4`.
  3. (Instr.) Interprétation de `return 4 * perimetre(3, 2) - (2 * 4 - 2) * 2`
  4. (Expr.) Evaluation de `4 * perimetre(3, 2) - (2 * 4 - 2) * 2`
  5. (Expr.) Evaluation de `4` $\longrightarrow$ `4`
  6. (Expr.) Evaluation de `perimetre(3, 2)`
  7. (Appel) Calcul de `perimetre`, `larg` vaut `2` et `long` vaut `3`.
  8. (Instr.) Interprétation de `return 2 * (2 + 3)`
  9. (Expr.) Evaluation de `2 * (2 + 3)` $\longrightarrow$ `10`
  10. (Retour) Valeur de retour de 7.: `10`.
  11. (Expr.) Simplification de 6.: `perimetre(3, 2)` vaut `10`

  ...

Suites d'Instructions

Idée

Décomposer un processus en tâches séquentielles.

Analogies

- Recettes de cuisine.
- Meubles suédois en kit / Jouets de construction.
- Patron de couture.

1 Brider une grosse poularde en entrée, la barder, la faire pocher à blanc. Lever les filets ; supprimer les os de l'estomac ; garnir l'intérieur de la poularde avec un appareil à mousse de volaille préalablement préparé de la façon suivante :
2 Mousse de volaille : Décortiquer les chairs d'un poulet reine poché à blanc et refroidi. Parer ces chairs et les piler au mortier en leur ajoutant un tiers de leur poids de foie gras cuit. Passer ce mélange au tamis fin ; le mettre dans une terrine ; le travailler en pleine glace en lui ajoutant 2 décilitres de gelée de volaille mi-prise assez réduite, et 3 décilitres de crème fouettée bien ferme.
3 Napper de sauce chaud-froid blanche les parties inférieures de la poularde farcie. Mettre cette dernière dans une coupe en cristal ovale, sur un fond de gelée solidifiée.
4 Garnir le dessus de la poularde avec les filets détaillés en aiguillettes, ces dernières légèrement arrondies sur un bout, décorées avec truffes et moitiés de pistaches mondées, et lustrées à la gelée. (Afin de bien égaliser le dressage de ces aiguillettes et de le rendre solide, pousser au cornet, sous chaque aiguillette, un mince cordon de mousse.) Lustrer la poularde à la gelée. Faire bien refroidir au rafraîchissoir.

Instructions en Python

- Juxtaposition verticale dans le corps d'une fonctions:

```
[instruction_1]
[instruction_2]
...
[instruction_n]
```

- Principe d'interprétation séquentiel
  1. on interpréte `[instruction_1]` entièrement.
  2. on interpréte `[instruction_2]` entièrement.
  3. ...
- Jusqu'ici, une seule instruction.
- `return` arrête la fonction.
  - séquence inutile (pour le moment).

Instruction d'affichage

- `print` permet d'afficher la valeur d'une expression à l'écran.
- Intérêt: déboguage, trouver à quel endroit, et de quelle manière, une fonction "se trompe".
- précisément, `print(expr1, expr2, ..., exprn)` affiche sur la sortie standard la valeur des expressions `expr1, ..., exprn` séparées par des espaces, avec un saut de ligne à la fin.

```
def h0(x : int) -> int:
    print(x +1)
    return x

=== Evaluation de : 'h0(42)' ===
43
42
=================================
```

- interaction avec la séquence:

```
def h1(x : int) -> int:
    return x
    print(x + 1)

=== Evaluation de : 'h1(42)' ===
42
=================================
```

Modèle simpliste d'ordinateur

- un processeur qui effectue des calculs
- une mémoire qui stocke des informations:
  - des résultats temporaires,
  - des fonctions,
  - des données.

Jusqu'ici:

- Utilisation du processeur pour:
  - évaluer des expressions,
  - interpréter des instructions.
- Utilisation de la mémoire pour:
  - stocker les arguments.
  - stocker les fonctions définies par l'utilisateur.

Paramètres en mémoire

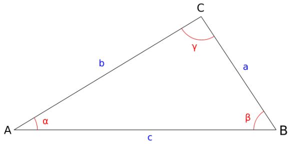
```
def perimetre(largeur : int, longueur : int) -> int:
    """ precondition : (longueur >= 0) and (largeur >= 0)
    precondition : longueur >= largeur
    retourne le perimetre du rectangle defini par sa largeur et sa longueur."""

    return (largeur + longueur) + (largeur + longueur)
```

- les paramètres `largeur` et `longueur` peuvent être considérés comme des cases mémoire.
  - contient une unique valeur, celle de l'argument.
  - est accessible en lecture.
  - est effacée à la fin de la fonction.

Evaluation de `perimetre(3, 2 * 2)`

- on met l'argument `3` dans le paramètre `largeur`
- on met l'argument `4` dans le paramètre `longueur`
- On interprète l'instruction `return` en évaluant l'expression,
- on accede à `largeur`, puis à `longueur`, puis à `largeur`, puis à `longueur`.

Aire du triangle



Objectif

Utiliser la mémoire pour plus que les arguments, de manière explicite.

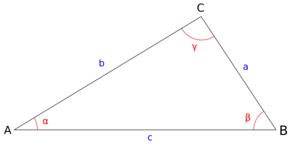
Problème

Calculer l'aire d'un triangle à partir des longueurs de ses trois côtés.

Contrat

`aire_triangle(3, 4, 5)` doit faire

Aire du triangle



Objectif

Utiliser la mémoire pour plus que les arguments, de manière explicite.

Problème

Calculer l'aire d'un triangle à partir des longueurs de ses trois côtés.

Contrat

`aire_triangle(3, 4, 5)` doit faire `6`.

Spécification

Méthode:

1. Que doit calculer la fonction ?
   - donne le nom
   - ici: aire d'un triangle à partir de ses côtés.
2. Quels sont les paramètres et leurs types ?
   - donne leur nombre, leurs noms, leurs types
   - ici: les trois côtés `a`, `b`, `c`,
   - ici: tous les trois de type `float`.
3. Quelle est valeur de retour de la fonction ?
   - donne le type de la sortie et la description de la fonction.
   - ici: l'aire du triangle, de type `float` (on utilise `/`).
4. Que doivent vérifier les paramètres ?
   - donne les precondition
   - ici: les longueurs sont positives,
   - ici: elles correspondent à un triangle (par exemple, pas `3`, `4`, `10`).

Spécification

Méthode:

1. Que doit calculer la fonction ?
   - donne le nom
   - ici: aire d'un triangle à partir de ses côtés.
2. Quels sont les paramètres et leurs types ?
   - donne leur nombre, leurs noms, leurs types
   - ici: les trois côtés `a`, `b`, `c`,
   - ici: tous les trois de type `float`.
3. Quelle est valeur de retour de la fonction ?
   - donne le type de la sortie et la description de la fonction.
   - ici: l'aire du triangle, de type `float` (on utilise `/`).
4. Que doivent vérifier les paramètres ?
   - donne les precondition
   - ici: les longueurs sont positives,
   - ici: elles correspondent à un triangle (par exemple, pas `3`, `4`, `10`).

```python
def aire_triangle(a : float, b : float, c : float) -> float :
    """ Precondition : (a>0) and (b>0) and (c>0)
    Predondition : 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."""
```

Algorithmique

Ensuite on doit trouver l'algorithme qui résout le problème.

Algorithme

- Programme indépendant du langage: suite d'instructions.
- Beaucoup d'algorithmes sont déjà connus:
  - *Euclide*, *Exponentiation rapide*, *Médiane en temps linéaire*, *Tri de Hoare*, *Ford-Fulkerson*, *Martelli-Montanari*.
- Trouver un algorithme est difficile.
  - imagination, créativité, vision mathématique, internet.
  - formellement: indécidable.

Ici:

Algorithmique

Ensuite on doit trouver l'algorithme qui résout le problème.

Algorithme

- Programme indépendant du langage: suite d'instructions.
- Beaucoup d'algorithmes sont déjà connus:
  - *Euclide*, *Exponentiation rapide*, *Médiane en temps linéaire*, *Tri de Hoare*, *Ford-Fulkerson*, *Martelli-Montanari*.
- Trouver un algorithme est difficile.
  - imagination, créativité, vision mathématique, internet.
  - formellement: indécidable.

Ici:

Formule d'Héron d'Alexandrie

1. Calculer le demi-périmètre du triangle: $s = \frac{a+b+c}{2}$
2. L'aire vaut $\sqrt{s(s-a)(s-b)(s-c)}$

Implémentation

Une première implémentation (naïve):

```
import math  # necessaire pour pouvoir utiliser la racine carree

def aire_triangle_naive(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."""

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

Problème

Implémentation

Une première implémentation (naïve):

```
import math  # necessaire pour pouvoir utiliser la racine carree

def aire_triangle_naive(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."""

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

Problème

- On calcule 4 fois le demi-périmètre.
- La fonction est difficile à lire.

Objectif: analogie avec la formule, calculer s 1 fois puis l'utiliser 4 fois.

Implémentation (II)

On utilise une case mémoire pour stocker le demi-périmètre.

```
import math  # necessaire pour pouvoir utiliser la racine carree

def aire_triangle(a : float, b : float, c : float) -> float :
    """ Precondition : (a>0) and (b>0) and (c>0)
    Predondition : 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."""

    s : float = (a + b + c) / 2

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

- un seul calcul du demi-périmètre, 4 utilisation.
- s est une variable locale à la fonction.
  - elle n'existe que dans la fonction.

Validation

- Jeu de test respectant les préconditions.
  - inégalités triangulaires: $a \leq b + c, \; b \leq a + c, \; c \leq a + b$.

```
# Jeu de tests (Etape 3)
assert aire_triangle(3, 4, 5) == 6.0
assert aire_triangle(13, 14, 15) == 84.0
assert aire_triangle(1, 1, 1) == math.sqrt(3 / 16)
assert aire_triangle(2, 3, 5) == 0.0  # c'est un triangle plat...
```

- Remarque: on peut changer les préconditions de la fonction:

  `precondition : les cotes a, b et c definissent bien un triangle.`

  devient

  `precondition : (a <= b + c) and (b <= a + c) and (c <= a + b)`
- Différence préconditions formelles/informelles:
  - LU1IN001: les deux sont acceptables.

Variables

Définition

Une variable est une case mémoire locale à une fonction.
Une variable est définie par:

1. un nom choisi par le programmeur.
2. un type de contenu: `int`, `float`, ...
3. une valeur correspondant à son contenu.

"Une variable, c'est un tiroir."

Manipulation de variables

- déclaration (Typ.): annoncer la présence d'une variable.
- initialisation (Instr.): premier contenu de la case mémoire.
  - on fait les deux en même temps: définition
- occurence au sein d'une expression (Expr.): utilisation (lecture) du contenu de la case.
- réaffectation (Instr.): mise à jour du contenu.

Définition de Variable

- Syntaxe : `var : type = expr`
- Typage nécessaire à son utilisation (*MrPython*).
- *MrPython*: inférence de type
  - vérifie que l'utilisation correspond au type déclaré.
- Instruction nécessaire à son existence.
  - `s : float = (a + b + c) / 2`
- Définitions obligatoire, en début de fonction, pour toutes les variables, sauf les variables d'itération (Cours 05).

Structure usuelle d'une fonction

```
def ma_fonction(param1 : T1, param2 : T2, ...) -> T:
    """partie 1 : preconditions et description
    ..."""

    # partie 2 : definition des variables

    v1 : U1 = expr1

    v2 : U2 = expr2

    ...

    vn : UN = exprN

    # partie 3 : implementation de l'algo
    instruction1
    instruction2
    ... etc ...
```

Occurence et Affectation

Occurence

- Expression permettant l'utilisation de la variable dans une expression.
- Syntaxe : `var`
  - 4 occurences de `s` dans `math.sqrt(s * (s - a) * (s - b) * (s - c))`
- Principe d'évaluation:
  - On évalue la variable par la valeur qu'elle contient.
    - au moment de l'évaluation.
- Remarque : une variable contient une valeur, pas un calcul.

Réaffectation

- Instruction qui modifie la valeur d'une variable.
- Syntaxe : `var = expr`
  - `=` n'est pas symétrique.
- Principe d'interprétation:
  1. On évalue `expr`.
  2. On remplace le contenu de `var` par la valeur de `expr`.

Représentation

```
def essai_var() -> int:

    n : int = 0

    m : int = 58

    n = m - 16
    m = m + 1
    return n + m
```

Représentation des variables par des tableaux:

1. apres la 1ere étape (définition de `n`):

variable	n
valeur	0

2. apres la 2eme étape (définition de `m`):

variable	n	m
valeur	0	58

3. apres la 3eme étape (réaffectation de `n`):

variable	n	m
valeur	42	58

   - Calcul de l'expression `m - 16`.

4. apres la 4eme étape (réaffectation de `m`):

variable	n	m
valeur	42	59

   - Calcul de l'expression `m + 1`.

Nommer les variables

- A la discrétion du programmeur.
  - en pensant aux relecteurs.
- Recommandations:
  - (Monde Réel) Doit être explicite.
    - utiliser, si nécessaire, des noms longs.
    - `demi_perimetre` plutôt que `s`.
  - (PEP8) mots en minuscules de lettres `a` à `z` séparés par des `_`
    - chiffres autorisés à la fin du nom.
    - Bon: `compteur`, `plus_grand_nombre`, `calcul1`, `min_liste1`
    - Mauvais: `Compteur`, `plusgrandnombre`, `1calcul`, `min_liste_1`
  - (PEP8) Même règle pour les noms de fonctions.
    - (LU1IN001) doit décrire le résultat.

Alternative

Définition

Instruction permettant de choisir entre deux séquences d'instructions selon la valeur d'une condition.

(aussi: *conditionnelle*, *branchement*)

- Fondamental en programmation (s'arrêter, faire des cas).
- Choix dans le calcul.

Valeur Absolue

Définie en mathématiques par $|x| = \begin{cases} x & \text{si } x \geq 0, \\ -x & \text{sinon.} \end{cases}$

- on fait un choix entre deux calculs (x ou $-x$) selon une condition ($x \geq 0$).
- On utiliser une alternative pour implémenter:

```
def valeur_absolue(x : float) -> float :
    """ retourne la valeur absolue de x."""
```

Alternative

- Syntaxe de l'instruction alternative:

```
if condition:
    consequent
else:
    alternant
```

  - `condition`: expression booléenne.
  - `consequent`: (séquence d') instruction(s)
  - `alternant`: (séquence d') instruction(s)
- erreurs fréquentes: indentations (*nesting*) et
- Principe d'interprétation:
  1. On évalue la `condition`
  2.
     - Si elle vaut **`True`**, on interprète le conséquent.
     - Si elle vaut **`False`**, on interprète l'alternant.

Valeur Absolue

```python
def valeur_absolue(x : float) -> float:
    """retourne la valeur absolue de x."""

    abs_x : float = 0    #  stockage de la valeur absolue, le choix de 0 pour
                         # l'initialisation est ici arbitraire

    if  x >= 0:
        abs_x = x  #consequent
    else:
        abs_x = -x # alternant

    return abs_x

# Jeu de tests
assert valeur_absolue(3) == 3
assert valeur_absolue(-3) == 3
assert valeur_absolue(1.5 - 2.5) == valeur_absolue(2.5 - 1.5)
assert valeur_absolue(0) == 0
assert valeur_absolue(-0) == 0
```

- résultat du calcul stocké dans une variable.
- contenu de la variable dépendant de la condition.

Valeur Absolue (II)

- Calcul de `valeur_absolue(3)`:
  1. définition

variable	abs_x
valeur	0

  2. la condition `3 >= 0` s'évalue en `True`, on choisit le conséquent.
  3. affectation

variable	abs_x
valeur	3

  4. On retourne la valeur de `abs_x`, c'est à dire `3`.
- Calcul de `valeur_absolue(-3)`:
  1. définition

variable	abs_x
valeur	0

  2. la condition `-3 >= 0` s'évalue en `False`, on choisit l'alternant.
  3. affectation

variable	abs_x
valeur	3

     car `-x` s'évalue en `3`.
  4. On retourne la valeur de `abs_x`, c'est à dire `3`.

Sortie anticipée

```python
def valeur_absolue2(x : float) -> float:
    """ retourne la valeur absolue de x."""

    if x >= 0:
        return x  # consequent
    else:
        return -x # alternant

# Jeu de tests
assert valeur_absolue2(3) == 3
assert valeur_absolue2(-3) == 3
assert valeur_absolue2(1.5 - 2.5) == valeur_absolue2(2.5 - 1.5)
assert valeur_absolue2(0) == 0
assert valeur_absolue2(-0) == 0
```

- On peut utiliser une instruction `return` comme conséquent ou alternant.
- On sort de la fonction "avant la fin".
- Efficacité légèrement meilleure.

Expressions Booléennes

- Expression de type `bool`.
- Deux valeurs possibles: `True` (vrai, $\top$) et `False` (faux, $\perp$).
- Expressions booléennes composées par des opérateurs:
  - Comparaisons de nombres `float * float ->bool`:
    $<$, `>, <=, >=, ==, !=`
  - Opérateurs logiques `bool * bool ->bool` et `bool ->bool`: `and`, `or`, `not`
- Ne pas confondre affectation (Instr.) et égalité (Expr.).

```
if i = 0:
    ...
```

Négation

- Prend l'opposé (dans les booléens) de son paramètre.
-
Valeur de `b`	**Valeur de** `not b`
`True`	`False`
`False`	`True`
- Principe d'évaluation de `not expr`
  - On évalue `b`
  -
    - si `b` vaut `True` on renvoie `False`,
    - sinon (`b` vaut `False`) on renvoie `True`.

Opérateurs Binaires

- Principe d'évaluation de `expr1 op expr2`:
  - On évalue `expr1` en `v1`.
  - On évalue `expr2` en `v2`.
  - On calcule une valeur dépendant de `v1` et `v2` (et de la sémantique de `op`).
- Fonctionne pour tous les opérateurs de comparaison.
- Conjonction (*et logique*):
  - `expr1 and expr2` vaut `True` quand les deux expressions valent `True`.
  - Principe d'évaluation de `expr1 and expr2`:

Opérateurs Binaires

- Principe d'évaluation de `expr1 op expr2`:
  - On évalue `expr1` en `v1`.
  - On évalue `expr2` en `v2`.
  - On calcule une valeur dépendant de `v1` et `v2` (et de la sémantique de `op`).
- Fonctionne pour tous les opérateurs de comparaison.
- Conjonction (*et logique*):
  - `expr1 and expr2` vaut **`True`** quand les deux expressions valent **`True`**.
  - Principe d'évaluation de `expr1 and expr2`:
    1. On évalue `expr1` en `v1`.
    2. Si `v1` vaut **`False`**, on renvoie **`False`** sans calculer `expr2`.
    3. Sinon on évalue `expr2` en `v2`.
    4. Si `v2` vaut **`False`**, on renvoie **`False`** sinon on renvoie **`True`**.
  - Calcul paresseux !

Opérateurs binaires (II)

- Disjonction (*ou logique*)
  - `expr1 or expr2` vaut `True` quand au moins une des deux expressions vaut `True`.
  - Principe d'évaluation de `expr1 or expr2`:
    1. On évalue `expr1` en `v1`.
    2. Si `v1` vaut `True`, on renvoie `True` sans calculer `expr2`.
    3. Sinon on évalue `expr2` en `v2`.
    4. Si `v2` vaut `True`, on renvoie `True` sinon on renvoie `False`.
  - Symétrique de `and`.
  - Calcul paresseux !
  - Utile pour l'efficacité ou les préconditions

```python
def est_divisible(n : int, d: int) -> bool:
    """ P: d != 0"""
    return (n == 0) or (n % d == 0)
```

  - `not` est prioritaire sur les opérateurs de comparaison.
    - `not True and False` vs. `not (True and False)`

Prédicats

Définition

Fonction qui renvoie un booléen.

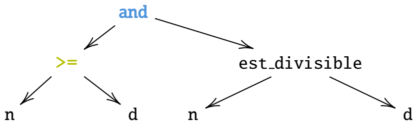
- une application d'un prédicat à des arguments (e.g. `est_divisible(12,4)`) est une expression booléenne (composée).
- on peut composer les opérateurs logiques et les prédicats pour obtenir des expressions booléennes complexes.
  - exemple: `(n >= d) and est_divisible (n, d)`

Prédicats

Définition

Fonction qui renvoie un booléen.

- une application d'un prédicat à des arguments (e.g. `est_divisible(12,4)`) est une expression booléenne (composée).
- on peut composer les opérateurs logiques et les prédicats pour obtenir des expressions booléennes complexes.
  - exemple: `(n >= d) and est_divisible (n, d)`



- les conditions des alternatives font souvent appel aux prédicats.

```
...
if (n >= d) and est_divisible (n, d):
    ...
```

Alternatives Multiples

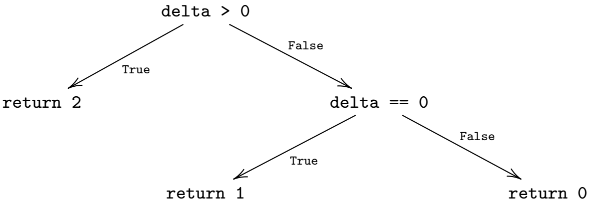
Problème

Définir la fonction `nb_solutions` qui, étant donné trois nombres `a`, `b` et `c`, renvoie le nombre de solutions de l'équation du second degré $a.x^2 + b.x + c = 0$.

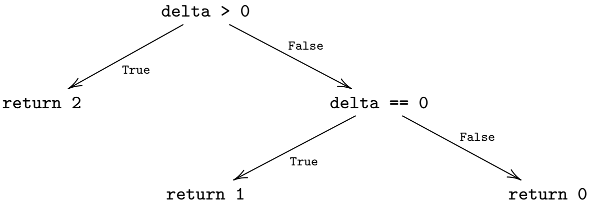
- On sait qu'il faut calculer le discriminant et discuter:
  - si le discriminant est strictement positif il y a 2 solutions,
  - si le discriminant est nul il y a 1 solution,
  - si le discriminant est négatif il n'y a aucune solution.
- On a un choix en trois cas.
- On imbrique deux alternatives:

```python
def nb_solutions(a : float, b : float, c : float) -> int :
    """ donne le nombre de solutions de l'equation a*x^2 + b*x + c = 0"""

    delta : float = b * b - 4 * a * c

    if delta > 0:
        return 2
    else:
        if delta == 0:
            return 1
        else:
            return 0
```

Alternatives Multiples



Alternatives Multiples



Alternative multiples:

```python
def nb_solutions2(a : float, b : float, c : float) -> int :
    """ donne le nombre de solutions de l'equation a*x^2 + b*x + c = 0"""

    delta : float = b * b - 4 * a * c

    if delta > 0:
        return 2
    elif delta == 0:
        return 1
    else:
        return 0
```

Alternatives Multiples

- On utilise l'alternative multiple, de syntaxe:

```
if condition1:
    consequent1
elif condition2:
    consequent2
elif ...
...
else:
   alternant
```

- Principe d'interprétation:
  1. On évalue la `condition1`
  2. - Si elle vaut **`True`**, on interprète uniquement le `consequent1`.
     - Si elle vaut **`False`**, on évalue la `condition2`.
  3. - Si elle vaut **`True`**, on interprète uniquement le `consequent2`.
     - Si elle vaut **`False`**, on évalue la `condition3`.
  4. ...on évalue la `condition42`.
     - Si elle vaut **`True`**, on interprète uniquement le `consequent42`.
     - Si elle vaut **`False`**, on interprète uniquement l'`alternant`.
- On n'évalue qu'un seul conséquent ou alternant.

Conditionnelle: Pas d'alternant

```
if cond:
    consequent
```

- Principe d'interprétation:
  1. On évalue la `condition`
  2. - Si elle vaut **`True`**, on interprète le conséquent.
- Utile pour faire quelque chose sous condition.
- "Sinon ne rien faire."

```
def valeur_absolue(x : float) -> float:

    abs : float = x

    if abs < 0:
        abs = -abs
    return abs
```

```
def valeur_absolue(x : float) -> float:

    if x < 0:
        return -x
    return x
```

Conditionnelle: Raccourcis

- On considère souvent les comparaisons comme une opération élémentaire pour la complexité.
  - la condition d'un `if` contient souvent une (ou plusieurs) comparaison(s).
  - on compare des fonctions selon le nombre de comparaisons qu'elles font sur des entrées similaires.
- Le branchement lui-même ajoute du temps de calcul.
- Eviter les alternatives, pour gagner du temps d'execution.

```python
if cond1:
    return True
else:
    return False
```

meilleur: `return cond1`

```python
if cond1:
    return False
else:
    return True
```

meilleur: `return not cond1`

```python
if cond1:
    s = True
else:
    s = False
```

meilleur: `s = cond1`

- Simplifier les conditions:

```python
if cond1 == True:
    ...
```

meilleur: `if cond1:`

```python
if cond1 == False:
    ...
```

meilleur: `if not cond1:`

- Sanctionné à l'examen.

Effets de bords

Définition

Un effet de bord est une instruction d'une fonction qui modifie un état (la mémoire, l'affichage) autre que la valeur de retour de la fonction.

- souvent son interprétation n'a pas d'effet direct sur le calcul.

- Affichage: `print` est un effet de bord, elle affiche sur la sortie standard.
- la modification de fichiers ("disque dur") est un effet de bord.
- Nécessaires, mais difficile à analyser.
  - idempotence des fonctions ?
- `print` fait un effet de bord: affichage à l'écran
  - utile pour connaître les valeurs intermédiaires des variables.
- Au programme du Cours 08 de 011 !

Valeurs Intermédiaires

```
def essai_var3(x : int) -> int:

    n : int = 0
    print("la valeur de n est:", format(n))

    m : int = x
    print("la valeur de n est:", format(n), "la valeur de m est:", format(m))

    n = m + x
    print("la valeur de n est:", format(n), "la valeur de m est:", format(m))
    m = n + 1
    print("la valeur de n est:", format(n), "la valeur de m est:", format(m))
    n = m + x
    print("la valeur de n est:", format(n), "la valeur de m est:", format(m))
    return n
```

- A utiliser en TME pour déboguer les fonctions.

Fonctions partielles

- primitive `print`:
  - utilisation courante: afficher des chaînes de caractères.
  - peut contenir des expressions de différents types.
  - la valeur de retour de `print` est

Fonctions partielles

- primitive `print`:
  - utilisation courante: afficher des chaînes de caractères.
  - peut contenir des expressions de différents types.
  - la valeur de retour de `print` est Rien

Fonctions partielles

- primitive `print`:
  - utilisation courante: afficher des chaînes de caractères.
  - peut contenir des expressions de différents types.
  - la valeur de retour de `print` est Rien (en `Python`: `None`).
  - le type de `None` est

Fonctions partielles

- primitive **`print`**:
  - utilisation courante: afficher des chaînes de caractères.
  - peut contenir des expressions de différents types.
  - la valeur de retour de **`print`** est Rien (en `Python`: `None`).
  - le type de `None` est `None`:
- Fonctions qui n'ont pas de valeur de retour:

```
def affiche_trois_fois(n : int) -> None:
    print(n)
    print(n)
    print(n)

assert affiche_trois_fois(10) == None
```

  - Est-ce vraiment des fonctions ?
    - Cours 08 : Procédures
- Plus tard dans l'UE, types optionnels
  - renvoyer soit un entier (quand ça "marche"), soit rien (quand ce n'est pas possible)

```
def racine(x : float) -> Optional[float]:
    if x >= 0:
        return math.sqrt(x)
```

Constantes Globales

- On peut affecter des constantes en dehors des fonctions ("globales").
  - elle doivent être déclarées.
- Ces constantes ne sont pas accessibles dans les fonctions.
- Ces constantes ne sont pas modifiables.
- Utiles pour les tests et les essais.
  - surtout avec des structures de données (cours 05-10).

```python
nombre : int = 42

def increm(x : int) -> int:
    return x + 1

assert increm(nombre) == 43

########################

def ajoute_n(x : int) -> int:
    return x + nombre # ERREUR

nombre = nombre + 1 # ERREUR
```

Modèle mémoire

- Mémoire est un espace indicé:
  - chaque "tiroir" a une taille et une adresse.
- une variable, c'est un nom pour l'adresse d'un tiroir,
  - une table de symboles lie noms et adresses.
- deux "zones" de mémoire:
  - le tas: où vivent les variables globales, les données, les objets, les fonctions (le code),
  - la pile: qui sert à l'execution de fonction.
    - contient les variables locales et les arguments,
    - durée de vie limitée,
    - cas des fonctions qui appellent d'autres fonctions
- En LU1IN002: modèle mémoire formel.

Expressivité et Terminaison

Expressivité

L'expressivité d'un langage, c'est l'ensemble des fonctions mathématiques qui peuvent être calculées par des fonctions informatiques (programmes) écrites dans ce langage.

- Jusqu'ici, trois instructions:
  - `return`,
  - affectation `=`,
  - alternative `if :` / `else :`.
- Expressivité limitée (calculatrice): primalité ?
- Implémentation de formules mathématiques (conditionnées).

Terminaison

Une fonction (informatique) `f` termine sur l'entrée `e` quand l'exécution de `f(e)` finit par s'arrêter. Elle termine quand elle termine sur toutes ses entrées.

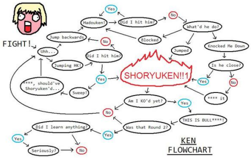
- Toutes nos fonctions terminent trivialement.

Que nous manque t-il ?

Objectif

Pouvoir répéter une action *aussi longtemps que nécessaire*.

- Analogie culinaire:
  - monter des blancs en neige,
  - cuire un gâteau.
- Répéter des actions similaires, potentiellement différentes.
- Comment exprimer aussi longtemps que nécessaire ?
- Terminaison ?



Calcul de la somme des premiers entiers

Problème

Calculer la somme des n premiers entiers.

Calcul de la somme des premiers entiers

Problème

Calculer la somme des n premiers entiers.

(*willing suspension of disbelief:* "Gauss n'a jamais existé: $\sum_{i=1}^{n} i = \frac{n.(n+1)}{2}$ est inconnue.")

Par exemple, si n vaut 5

```
def somme5() -> int:
    """retourne la somme des 5 premiers entiers naturels."""

    return 1 + 2 + 3 + 4 + 5

# Test
assert somme5() == 15
```

Calcul de la somme des premiers entiers

Problème

Calculer la somme des n premiers entiers.

(*willing suspension of disbelief:* "Gauss n'a jamais existé: $\sum_{i=1}^{n} i = \frac{n.(n+1)}{2}$ est inconnue.")

Par exemple, si n vaut 5

```python
def somme5() -> int:
    """retourne la somme des 5 premiers entiers naturels."""

    return 1 + 2 + 3 + 4 + 5

# Test
assert somme5() == 15
```

- Malaise:
  - solution spécifique à $n = 5$.
    - définir une fonction pour chaque entier.
  - Ecriture fastidieuse quand $n = 100000$.
- On voudrait:
  - une définition générale `def somme(n : int) ->int`,

Calcul de la somme des premiers entiers (II)

- Calculs répétitifs: nombre d'étapes n'est pas fixe.
- Math: formule générale $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n$.
  - n est paramètre de la formule (pas i).
- A la main, approche itérative:
  - la somme s vaut 0 initialement,
  - on ajoute le premier entier 1, s vaut 1,
  - on ajoute l'entier suivant 2, s vaut 3,
  - on ajoute l'entier suivant 3, s vaut 6,
  - on ajoute l'entier suivant 4, s vaut 10,
  - on ajoute l'entier suivant 5, s vaut 15,
  - on a atteint la borne 5, on s'arrête et la somme vaut 15.

```python
def somme_ite5() -> int:
    """retourne la somme des 5 premiers entiers naturels."""

    s : int = 0 # valeur temporaire de la somme

    s = s + 1
    s = s + 2
    s = s + 3
    s = s + 4
    s = s + 5
    return s
```

Calcul de la somme des premiers entiers (III)

- Même traitement à chaque étape.
  - Une instruction: affecter une nouvelle valeur à la variable `s`.
- Idée:
  - une variable i initialisée à 1 pour représenter, successivement les entiers de 1 à n.
  - une variable s initialisée à 0 pour représenter la somme des entiers jusqu'à l'entier courant.
  - a chaque étape (en tout n fois):
    - affecter à s sa valeur courante augmentée de i,
    - faire passer i à l'entier suivant (incrémentation).
  - arrêter quand on a fait n étapes avec n paramètre.
- Outil fourni dans tout (?) langage de prog.: les boucles.

Calcul de la somme des premiers entiers (IV)

- Boucle `while`: répète un bloc d'instruction tant qu'une certaine condition (expression booléenne) est vérifiée.
- Ici, principe de répétition:
  - répéter tant que `i <= 5` (condition) le bloc d'instructions suivant:
    1. Instr. ajouter *s* le contenu de *i* (`s = s + i`).
    2. Instr. incrémenter *i* (`i = i + 1`).

```python
def somme_while5() -> int:
    """retourne la somme des 5 premiers entiers naturels."""

    i : int = 1 # entier courant

    s : int = 0 # la somme cumulee

    while i <= 5:
        s = s + i
        i = i + 1

    return s

# Test
assert somme_while5() == 1 + 2 + 3 + 4 + 5
```

- Terminaison ?: `i` vaut `6` au 5eme tour, condition fausse, on sort.

Syntaxe du *while*

- Syntaxe:

```
while cond:
    instruction_1
    instruction_2
    ...
    instruction_n
```

  - `cond` est la condition de la boucle, c'est une expression booléenne
  -

```
    instruction_1
    instruction_2
    ...
    instruction_n
```

    est le corps de la boucle (ce qui est répété).
  - le corps est défini par l'indentation:

```
while cond
    instruction_1
    instruction_2
    ...
    instruction_n
instruction_n1
```

    ici, `instruction_n1` ne fait par partie du corps de la boucle, elle n'est pas répétée, elle est exécutée en sortie de la boucle.

Calcul de la somme des premiers entiers (V)

- Grâce au `while`: écriture synthétique de `somme5`.
- Généraliser la somme: n comme paramètre, pour remplacer `5`.

```
def somme_entiers(n : int) -> int:
    """Precondition: n >= 1
    retourne la somme des n premiers entiers naturels."""

    i : int = 1 # entier courant, en commencant par 1

    s : int = 0 # la somme cumulee

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

    return s
```

Exercices

Somme des carrés.

Donner une fonction `somme_carres` qui prend en entrée un entier naturel `n` et renvoie la somme des carrées des entiers de 0 jusque `n`.

- Spécification:

Exercices

Somme des carrés.

Donner une fonction `somme_carres` qui prend en entrée un entier naturel `n` et renvoie la somme des carrées des entiers de 0 jusque `n`.

- Spécification: `def somme_carres(n : int) ->int`
- Précondition:

Exercices

Somme des carrés.

Donner une fonction `somme_carres` qui prend en entrée un entier naturel `n` et renvoie la somme des carrées des entiers de 0 jusque `n`.

- Spécification: `def somme_carres(n : int) ->int`
- Précondition: `n >= 0`
- Algorithme:

Exercices

Somme des carrés.

Donner une fonction `somme_carres` qui prend en entrée un entier naturel `n` et renvoie la somme des carrées des entiers de 0 jusque `n`.

- Spécification: `def somme_carres(n : int) ->int`
- Précondition: `n >= 0`
- Algorithme: ajouter incrémentalement le carré de chaque entier, en partant de `0` et en s'arrêtant à `n`
- Implémentation:

Exercices

Somme des carrés.

Donner une fonction `somme_carres` qui prend en entrée un entier naturel `n` et renvoie la somme des carrées des entiers de 0 jusque `n`.

- Spécification: `def somme_carres(n : int) ->int`
- Précondition: `n >= 0`
- Algorithme: ajouter incrémentalement le carré de chaque entier, en partant de `0` et en s'arrêtant à `n`
- Implémentation:

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

  i : int = 0
  s : int = 0

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

- Validation:

Somme des carrés.

Donner une fonction `somme_carres` qui prend en entrée un entier naturel `n` et renvoie la somme des carrées des entiers de 0 jusque `n`.

- Spécification: `def somme_carres(n : int) ->int`
- Précondition: `n >= 0`
- Algorithme: ajouter incrémentalement le carré de chaque entier, en partant de `0` et en s'arrêtant à `n`
- Implémentation:

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

  i : int = 0
  s : int = 0

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

- Validation:

```
#Test
assert somme_carres(4) == 30
```

Exercices (II)

Somme des entiers impairs.

Donner une fonction `somme_impairs` qui prend en entrée un entier naturel `n` et renvoie la somme des entiers impairs compris entre 0 et `n` (inclus).

- Spécification:

Exercices (II)

Somme des entiers impairs.

Donner une fonction `somme_impairs` qui prend en entrée un entier naturel `n` et renvoie la somme des entiers impairs compris entre 0 et `n` (inclus).

- Spécification: `def somme_impairs(n : int) ->int`
- Précondition:

Exercices (II)

Somme des entiers impairs.

Donner une fonction `somme_impairs` qui prend en entrée un entier naturel `n` et renvoie la somme des entiers impairs compris entre 0 et `n` (inclus).

- Spécification: `def somme_impairs(n : int) ->int`
- Précondition: `n >= 0`
- Algorithme:

Exercices (II)

Somme des entiers impairs.

Donner une fonction `somme_impairs` qui prend en entrée un entier naturel `n` et renvoie la somme des entiers impairs compris entre 0 et `n` (inclus).

- Spécification: `def somme_impairs(n : int) ->int`
- Précondition: `n >= 0`
- Algorithme:
  - parcourir incrémentalement les entiers, partant de `0` et en s'arrêtant à `n`, et n'ajouter que ceux impairs.
  - parcourir de 2 en 2, en partant de `1` et en s'arrêtant à `n`.
  - parcourir incrémentalement les entiers de `0` à `(n - 1) // 2`, et ajouter le successeur de leur double à chaque étape.
- Implémentation:

```python
def somme_impairs1(n : int) -> int:
    """Precondition : n >= 0 """

    i : int = 0
    s : int = 0
    while i <= n:
        if i % 2 == 1:
            s = s + i
        i = i + 1
    return s
```

```python
def somme_impairs2(n : int) -> int:
    """Precondition : n >= 0 """
    i : int = 1
    s : int = 0
    while i <= n:
        s = s + i
        i = i + 2
    return s
```

```python
def somme_impairs3(n : int) -> int:
    """Precondition : n >= 0 """
    i : int = 0
    s : int = 0
    while i <= (n - 1) // 2:
        s = s + 2 * i + 1
        i = i + 1
    return s
```

- Validation:

Exercices (II)

Somme des entiers impairs.

Donner une fonction `somme_impairs` qui prend en entrée un entier naturel `n` et renvoie la somme des entiers impairs compris entre 0 et `n` (inclus).

- Spécification: `def somme_impairs(n : int) −>int`
- Précondition: `n >= 0`
- Algorithme:
  - parcourir incrémentalement les entiers, partant de `0` et en s'arrêtant à `n`, et n'ajouter que ceux impairs.
  - parcourir de 2 en 2, en partant de `1` et en s'arrêtant à `n`.
  - parcourir incrémentalement les entiers de `0` à `(n − 1) // 2`, et ajouter le successeur de leur double à chaque étape.
- Implémentation:

```
def somme_impairs1(n : int) −> int:
    """Precondition : n >= 0 """

    i : int = 0
    s : int = 0
    while i <= n:
        if i % 2 == 1:
            s = s + i
        i = i + 1
    return s
```

```
def somme_impairs2(n : int) −> int:
    """Precondition : n >= 0 """
    i : int = 1
    s : int = 0
    while i <= n:
        s = s + i
        i = i + 2
    return s
```

```
def somme_impairs3(n : int) −> int:
    """Precondition : n >= 0 """
    i : int = 0
    s : int = 0
    while i <= (n − 1) // 2:
        s = s + 2 * i + 1
        i = i + 1
    return s
```

- Validation:`assert somme_impairs(5) == 9`

Exercices (III)

Racine cubique approchée.

Donner une fonction `racine_cubique_entiere` qui prend en entrée un entier naturel `n` et renvoie la partie entière de sa racine cubique.

- Spécification:

Exercices (III)

Racine cubique approchée.

Donner une fonction `racine_cubique_entiere` qui prend en entrée un entier naturel `n` et renvoie la partie entière de sa racine cubique.

- Spécification: `def racine_cubique_entiere(n : int) ->int:`
- Hypothèse:

Exercices (III)

Racine cubique approchée.

Donner une fonction `racine_cubique_entiere` qui prend en entrée un entier naturel `n` et renvoie la partie entière de sa racine cubique.

- Spécification: `def racine_cubique_entiere(n : int) ->int:`
- Hypothèse: `n >= 0`
- Algorithme:
  - Problème: pas de primitive pour faire directement la racine cubique.

Exercices (III)

Racine cubique approchée.

Donner une fonction `racine_cubique_entiere` qui prend en entrée un entier naturel `n` et renvoie la partie entière de sa racine cubique.

- Spécification: `def racine_cubique_entiere(n : int) ->int:`
- Hypothèse: `n >= 0`
- Algorithme:
  - Problème: pas de primitive pour faire directement la racine cubique.
  - Solution: on parcourt incrémentalement tous les entiers et on les élève au cube, on s'arrête quand on dépasse `n`.
  - Différence: on ne sait pas à l'avance combien de tours de boucle on va faire.
- Implémentation et Validation:

```python
def racine_cubique_entiere(n : int) -> int:
    """Precondition : n >= 0 """
    racine : int = 0
    while (racine ** 3) <= n:
        racine = racine + 1
    return racine - 1

assert racine_cubique_entiere(30) == 3
```

Exercices (III)

Racine cubique approchée.

Donner une fonction `racine_cubique_entiere` qui prend en entrée un entier naturel `n` et renvoie la partie entière de sa racine cubique.

- Spécification: `def racine_cubique_entiere(n : int) ->int:`
- Hypothèse: `n >= 0`
- Algorithme:
  - Problème: pas de primitive pour faire directement la racine cubique.
  - Solution: on parcourt incrémentalement tous les entiers et on les élève au cube, on s'arrête quand on dépasse `n`.
  - Différence: on ne sait pas à l'avance combien de tours de boucle on va faire.
- Implémentation et Validation:

```
def racine_cubique_entiere(n : int) -> int:
    """Precondition : n >= 0 """
    racine : int = 0
    while (racine ** 3) <= n:
        racine = racine + 1
    return racine - 1

assert racine_cubique_entiere(30) == 3
```

- `return int(n ** (1 / 3))` fonctionne ... (intérêt pédagogique nul)

Principe d'interprétation du `while`

Pour interpréter

```
while cond:
    instruction_1
    ...
    instruction_n
instruction_apres_1
...
```

1. on évalue `cond`
2. si la valeur de `cond` n'est pas `False`, on interprète en entier:

   ```
   instruction_1
   ...
   instruction_n
   ```

   et on revient en 1.
3. si la valeur de `cond` est `False`, on sort de la boucle et on interprète la suite:

   ```
   instruction_apres_1
   ...
   ```

Simulation de boucle

Tables de simulation

1. Fixer les valeurs des paramètres (on simule sur un exemple précis)
2. Fixer les valeurs des variables non modifiées par la boucle.
3. Créer un tableau avec:
   3.1 une colonne tour de boucle,
   3.2 une colonne par variable modifiée par la boucle.
4. Remplir une ligne *entrée* avec les valeurs avant la boucle.
5. Décider s'il y a un tour de boucle en évaluant la condition.
6. Si oui, remplir une nouvelle ligne avec les valeurs en fin de tour.
7. Sinon, on écrit *(sortie)* au dernier tour.

Simulation de boucle: Exemple

```python
def somme_entiers(n : int) -> int:
    """Precondition: n >= 1
    retourne la somme des n premiers entiers naturels."""

    i : int = 1 # entier courant, en commencant par 1

    s : int = 0 # la somme cumulee

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

    return s
```

somme_entiers(5)

Simulation de boucle: Exemple

```
def somme_entiers(n : int) -> int:
    """Precondition: n >= 1
    retourne la somme des n premiers entiers naturels."""

    i : int = 1 # entier courant, en commencant par 1

    s : int = 0 # la somme cumulee

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

    return s
```

somme_entiers(5)

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

Utilisation de print

- `print` permet de tracer (obtenir une trace) des boucles,.
- on obtient exactement une simulation.

```python
def somme_entiers_tracee(n : int) -> int:
    """Precondition: n >= 1
    retourne la somme des n premiers entiers naturels."""

    i : int = 1 # compteur
    s : int = 0 # somme

    print("====================")
    print("s en entree vaut ", s)
    print("i en entree vaut ", i)
    while i <= n:
        s = s + i
        i = i + 1
        print("--------------------")
        print("s apres le tour vaut ", s)
        print("i apres le tour vaut ", i)
    print("--------------------")
    print("sortie")
    print("====================")

    return s
```

Suites récursives

Définition

Une suite récursive $(u_n)_{n \in \mathbb{N}}$ est définie par un premier terme k et une fonction de récursion f. On note:

$$\begin{cases} u_0 & = & k \\ u_{n+1} & = & f(u_n) \quad \text{pour } n \in \mathbb{N} \end{cases}$$

Exemple

$(u_n)_{n \in \mathbb{N}}$ définie par $\begin{cases} u_0 & = & 7 \\ u_{n+1} & = & 2 * u_n + 3 \quad \text{pour } n \in \mathbb{N} \end{cases}$

- Objectif: définir une fonction `valeur_u(n)` renvoyant la valeur du `n`-eme terme de la suite u_n donnée en exemple.
- problème similaire au précédent:
  - boucle **`while`** avec un compteur `i` et une accumulation `u`.

Suites Récursives (II)

```
def suite_u(n : int) -> int:
    """Precondition: n >= 0
    retourne la valeur au rang n de la suite U."""

    u : int = 7  # valeur au rang 0

    i : int = 0  # initialement rang 0

    while i < n:
        u = 2 * u + 3
        i = i + 1

    return u
```

Simulation `suite_u(6)`

Suites Récursives (II)

```python
def suite_u(n : int) -> int:
    """Precondition: n >= 0
    retourne la valeur au rang n de la suite U."""

    u : int = 7   # valeur au rang 0

    i : int = 0   # initialement rang 0

    while i < n:
        u = 2 * u + 3
        i = i + 1

    return u
```

Simulation suite_u(6)

tour de boucle	variable u	variable i
entrée	7	0
1	17	1
2	37	2
3	77	3
4	157	4
5	317	5
6 (sortie)	637	6

Suites Récursives (III)

- Généraliser: définir `suite_rec(n,f,k)` qui renvoie le `n`-ième terme de la suite de premier terme `k` et de fonction de récursion `f`.
- Difficulté: fonction de type `Callable[[int], int]` en paramètre.
- Ordre supérieur:
  - fonctions comme paramètre ou résultat de fonction
  - pas au programme de LU1IN001 (système de types d'ordre 1).
  - style de programmation fonctionnelle (Cours 11, LU2IN019, LI101).
- Présent dans l'informatique moderne (par exemple, dans le Web).

```python
def suite_rec(n : int, f : Callable[[int], int], k : int):
    """ Precondition: n >= 0
    retourne la valeur au rang n de la suite recursive
    de premier terme k et de fonction de recursion f."""

    u : int = k  # valeur au rang 0
    i : int = 0  # initialement rang 0

    while i < n:
        u = f(u)
        i = i + 1
    return u
```

Somme et produit des termes d'une suite

Objectif

Calcul des sommes et produits partiels des termes d'une suite.

Suite dyadique:

$\forall n \in \mathbb{N},\ u_n = \frac{1}{2^n}$

$\forall n \in \mathbb{N},\ S_n = \sum_{k=0}^{n} u_k = \sum_{k=0}^{n} \frac{1}{2^k}$

Somme et produit des termes d'une suite

Objectif

Calcul des sommes et produits partiels des termes d'une suite.

Suite dyadique:

$\forall n \in \mathbb{N},\ u_n = \frac{1}{2^n}$

$\forall n \in \mathbb{N},\ S_n = \sum_{k=0}^{n} u_k = \sum_{k=0}^{n} \frac{1}{2^k}$

```
def somme_partielle_u(n : int) -> float:
    """ Precondition: n >= 0
    retourne le n-ieme terme de la somme partielle  :
    1 + 1/2 + 1/4 + ... + (1/2)^n"""

    s : float = 0.0  # la somme vaut 0 initialement
    k : int = 0    # on commence au rang 0

    while k <= n:
        s = s + ((1/2) ** k)
        k = k + 1
    return s
```

Somme et produit des termes d'une suite (II)

Factorielle:
$\forall n \in \mathbb{N}^*, \; n! = \prod_{k=1}^{n} k$

Factorielle:

$\forall n \in \mathbb{N}^*,\ n! = \prod_{k=1}^{n} k$

```
def factorielle(n : int) -> int:
    """Precondition : n > 0
    retourne le produit factoriel n!"""

    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
```

Somme et produit des termes d'une suite (III)

- Objectif; calculer la somme des n-premiers termes d'une suite récursive à partir de son élément initial et de sa fonction de récursion.
- $S_n = \sum_{k=0}^{n} u_k = u_0 + f(u_0) + f(f(u_0)) + \dots$

Somme et produit des termes d'une suite (III)

- Objectif; calculer la somme des n-premiers termes d'une suite récursive à partir de son élément initial et de sa fonction de récursion.
- $S_n = \sum_{k=0}^{n} u_k = u_0 + f(u_0) + f(f(u_0)) + \ldots$

```
def somme_suite_rec(n : int, f : Callable[[float], float], k : float):
    """ Precondition: n >= 0
    renvoie la valeur de la somme partielle des n premiers termes
    de la suite recursive de premier terme k
    et de fonction de recursion f"""

    i : int = 0 # iterateur
    u : int = k # premier terme de la suite
    s : int = k # somme accumulee

    while i < n:
        u = f(u)
        s = s + u
        i = i + 1
    return s
```

Calcul du PGCD

Problème

Calculer le plus grand commun diviseur de deux entiers positifs.

Méthode standard

- `pgcd` doit calculer le pgcd de ses paramètres.
- deux paramètres `a` et `b`, entiers tels que `a>= b >= 0`.
- résultat est un entier.
- ```python
  def pgcd(n : int, m : int) -> int:
      """Precondition: n >= m > 0
      Retourne le plus grand commun diviseur de n et m."""
  ```
- Comment calculer le résultat ?
  - Trouver un algorithme pour résoudre le problème.
  ```

Calcul du PGCD: Rappels

- si $(k, n) \in \mathbb{N}^2$, k divise n s'il existe $m \in \mathbb{N}$ tel que $k.m = n$.
  - 3 divise 12 (car $3.4 = 12$).
  - 5 ne divise pas 12.
  - 42 divise 0 (car $42.0 = 0$).
- l'ensemble des diviseurs de $n \in \mathbb{N}$, noté $\texttt{div}(n)$, est l'ensemble des entiers de $\mathbb{N}$ qui divisent n.
  - $\texttt{div}(12) = \{1, 2, 3, 4, 6, 12\}$
  - $\texttt{div}(9) = \{1, 3, 9\}$
  - $\texttt{div}(13) = \{1, 13\}$
  - $\texttt{div}(0) = \mathbb{N}$
- l'ensemble des diviseurs communs de $n \in \mathbb{N}$ et de $m \in \mathbb{N}$, noté $\texttt{div}(n, m)$, est l'intersection des diviseurs de n et m (i.e. $\texttt{div}(n) \cap \texttt{div}(m)$)
  - $\texttt{div}(12, 9) = \{1, 3\}$
  - $\texttt{div}(12, 0) = \{1, 2, 3, 4, 6, 12\}$
- le *pgcd* de $n \in \mathbb{N}^*$ et de $m \in \mathbb{N}$, noté $\texttt{pgcd}(n, m)$, est le plus grand diviseur commun à n et m (i.e. $\texttt{max}(\texttt{div}(n, m))$)
  - $\texttt{pgcd}(12, 9) = 3$
  - $\texttt{pgcd}(12, 0) = 12$

Solution Naïve

- Utilise les ensembles (Cours 09) et les compréhensions (Cours 10)

```python
def diviseurs(n : int) -> Set[int]:
    """Precondition : n > 0"""
    return {k for k in range(1, n + 1) if n % k == 0}

def max_ensemble(E : Set[int]) -> int:
    """Precondition: E != set()
    Precondition: les elements de E sont positifs"""
    m : int = -1
    e : int
    for e in E:
        if e > m:
            m = e
    return m

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

    if m == 0:
        return n
    else:
        return max_ensemble(diviseurs(n) & diviseurs(m))

assert pgcd_naif(12, 9) == 3
```

- Meilleur algorithme ?

Algorithme d'Euclide

- Soit $n \in \mathbb{N}^*$ et $m \in \mathbb{N}$, la division euclidienne de n par m est l'unique couple $(q, r) \in \mathbb{N} \times [\![0, n-1]\!]$ tel que $n = q.m + r$
  - q est le quotient, obtenu avec `n // m`,
  - r est le reste, obtenu avec `n % m`,
  - avec 12 et 9 on a $(1, 3)$ car $12 = 1.9 + 3$
  - avec 12 et 6 on a $(2, 0)$ car $12 = 2.6 + 0$
- Propriété: Si (q, r) est la division euclidienne de n par m, alors $\mathtt{pgcd}(n, m) = \mathtt{pgcd}(m, r)$.
- Algorithme d'Euclide: Pour calculer $\mathtt{pgcd}(n, m)$:
  1. si m est 0, le pgcd est n,
  2. sinon
     2.1 on calcule r le reste de la division euclidienne de m par n
     2.2 on calcule $\mathtt{pgcd}(m, r)$. (récursion)
- Terminaison: on sait que $r < m$, donc "quelque chose" (ici la somme des deux nombres) décroît à chaque étape.

Algorithme d'Euclide: Exemples

- le pgcd de 56 et 42 est le pgcd de 42 et 14 $(56 = 42 * 1 + 14)$
- le pgcd de 42 et 14 est le pgcd de 14 et 0 $(42 = 14 * 3 + 0)$
- le pgcd de 14 et 0 est 14.

le pgcd de 56 et 42 est 14.

- le pgcd de 4199 et 1530 est le pgcd de 1530 et 1139 $(4199 = 1530 * 2 + 1139)$
- le pgcd de 1530 et 1139 est le pgcd de 1139 et 391 $(1540 = 1139 * 1 + 391)$
- le pgcd de 1139 et 391 est le pgcd de 391 et 357 $(1139 = 391 * 2 + 357)$.
- le pgcd de 391 et 357 est le pgcd de 357 et 34 $(391 = 357 * 1 + 34)$.
- le pgcd de 357 et 34 est le pgcd de 34 et 17 $(357 = 34 * 10 + 17)$.
- le pgcd de 34 et 17 est le pgcd de 17 et 0 $(34 = 17 * 10 + 0)$.
- le pgcd de 17 et 0 est 17.

le pgcd de 4199 et 1530 est 17.

Algorithme d'Euclide: Implémentation

```
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""
```

- Variables:

Algorithme d'Euclide: Implémentation

```python
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""
```

- Variables: deux variables `d` et `r` pour stocker les deux nombres à chaque étape.
  - elles contiennent initialement `n` et `m`.
- Condition de la boucle:

Algorithme d'Euclide: Implémentation

```python
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""
```

- Variables: deux variables `d` et `r` pour stocker les deux nombres à chaque étape.
  - elles contiennent initialement `n` et `m`.
- Condition de la boucle: continuer à faire des divisions euclidiennes tant que `r` est différent de `0`
  - sinon, le résultat est `d`
- Corps de la boucle:

Algorithme d'Euclide: Implémentation

```python
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""
```

- Variables: deux variables `d` et `r` pour stocker les deux nombres à chaque étape.
  - elles contiennent initialement `n` et `m`.
- Condition de la boucle: continuer à faire des divisions euclidiennes tant que `r` est différent de `0`
  - sinon, le résultat est `d`
- Corps de la boucle: mettre `d % r` dans `r` et `r` dans `d`.
  - ```python
        r = d % r
        d = r
    ```
    ```

Algorithme d'Euclide: Implémentation

```python
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""
```

- Variables: deux variables `d` et `r` pour stocker les deux nombres à chaque étape.
  - elles contiennent initialement `n` et `m`.
- Condition de la boucle: continuer à faire des divisions euclidiennes tant que `r` est différent de `0`
  - sinon, le résultat est `d`
- Corps de la boucle: mettre `d % r` dans `r` et `r` dans `d`.
  -
    ```python
        r = d % r
        d = r
    ```
  - Problème: instructions exécutées en séquence (`r` change)

Algorithme d'Euclide: Implémentation

```
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""
```

- Variables: deux variables `d` et `r` pour stocker les deux nombres à chaque étape.
  - elles contiennent initialement `n` et `m`.
- Condition de la boucle: continuer à faire des divisions euclidiennes tant que `r` est différent de `0`
  - sinon, le résultat est `d`
- Corps de la boucle: mettre `d % r` dans `r` et `r` dans `d`.
  - ```
        r = d % r
        d = r
    ```
  - Problème: instructions exécutées en séquence (`r` change)
  - Solution: variable temporaire (pour la future valeur de `r`):

```
    temp = d % r
    d = r
    r = temp
```
```

Algorithme d'Euclide: Implémentation

```python
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""
```

- Variables: deux variables `d` et `r` pour stocker les deux nombres à chaque étape.
  - elles contiennent initialement `n` et `m`.
- Condition de la boucle: continuer à faire des divisions euclidiennes tant que `r` est différent de `0`
  - sinon, le résultat est `d`
- Corps de la boucle: mettre `d % r` dans `r` et `r` dans `d`.
  - ```python
        r = d % r
        d = r
    ```
  - Problème: instructions exécutées en séquence (`r` change)
  - Solution: variable temporaire (pour la future valeur de `r`):

    ```python
        temp = d % r
        d = r
        r = temp
    ```
  - Cours 07: `d, r = r, d % r`
    ```

Algorithme d'Euclide: Implémentation (II)

```python
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""

    d : int = n
    r : int = m
    temp :int = 0   #  variable temporaire

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

Simulation de `pcgd(56, 42)`

Algorithme d'Euclide: Implémentation (II)

```
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""

    d : int = n
    r : int = m
    temp :int = 0   #  variable temporaire

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

Simulation de `pcgd(56, 42)`

tour de boucle	variable temp	variable q	variable r
entrée	0	56	42
1	14	42	14
2 (sortie)	0	14	0

Boucles imbriquées: couples d'entiers

Problème

Pour un entier positif n fixé, combien y existe t'il de couples d'entiers (i, j) tels que $i + j$ soit divisible par 3.

- pour $n = 0$ on en a 1: $(0, 0)$.
- pour $n = 1$ on en a 1: $(0, 0)$.
- pour $n = 2$ on en a 3: $(0, 0), (1, 2), (2, 1)$.

- Algorithme: impossible avec une boucle.
  - il faut faire varier i et j indépendamment.
  - pour chaque valeur de i, on parcourt toutes les valeurs possibles de j
  - espace quadratique
- Solution: boucles imbriquées.

Boucles imbriquées: couples d'entiers (II)

```python
def nombre_couples(n : int) -> int:
    """Precondition : n >= 0
    calcule le nombre de couple (i,j) tels que i.j est divisible par 3"""

    i : int = 0
    j : int = 0
    nb : int = 0

    while i <= n:
        j = 0
        while j <= n:
            if (i + j) % 3 == 0:
                nb = nb + 1
            j = j +1
        i = i + 1
    return nb
```

- l'indentation est cruciale,
- on ne doit pas oublier de remettre `j` à `0`

Boucles imbriquées: couples d'entiers (III)

Simulation de `nombre_couples(2)`

tour de boucle externe	tour de boucle interne	variable i	variable j	variable nb
entrée	-	0	0	0
1	entrée	0	0	0
1	1	0	1	1
1	2	0	2	1
1	3 (sortie)	0	3	1
2	entrée	1	0	1
2	1	1	1	1
2	2	1	2	1
2	3 (sortie)	1	3	2
3	entrée	2	0	2
3	1	2	1	2
3	2	2	2	3
3	3 (sortie)	2	3	3
3 (sortie)	-	3	3	3

- Une colonne par boucle, classées de l'extérieur vers l'intérieur.
- Simulation multiples : pas au programme des examens.
  - mais les boucles imbriquées, oui.
- Facilement traçable.

Zéro d'une fonction sur intervalle (I)

Problème

Décider si une fonction des entiers f s'annule sur l'intervalle entier $[\![a; b]\!]$.

Méthode standard

- la fonction `annule` doit décider si une fonction est égale à `0.0` sur un entier `x` compris entre deux bornes.
- trois arguments: une fonction `f` de type `Callable[[int], float]`, une borne inférieure `a` entière et une borne supérieure `b` entière.
- un résultat booléen.
- Algorithme: utiliser une boucle pour calculer successivement toutes les valeurs de `f` sur les entiers entre `a` et `b`
  - l'itérateur `x` va commencer à `a` puis être incrémenté successivement jusque valoir `b`.

Zéro d'une fonction sur un intervalle (II)

```
def annulation(a : int, b : int, f : Callable[[int], float]) -> bool:
    """ Precondition : a <= b
    Retourne True si la fonction f  s'annule sur l'intervalle [a;b]."""

    x : int = a  # element courant, au debut de l'intervalle

    while (x <= b):
        if f(x) == 0.0:
            return True  # la fonction s'annule !
        else:  # sinon on continue avec l'element suivant
            x = x + 1

    return False  # on sait ici que la fonction ne s'annule pas
```

- Il faut des fonctions utilisables en argument, par exemple:

```
def parabole(x : float) -> float:
    """ Calcule la valeur de X^2+X-6    """

    return x * x + x - 6

assert annulation(0, 10, parabole) == True
assert annulation(10, 20, parabole) == False
```

- Sous-typage contravariant avec l'argument:
  Comme on a `float` $\subseteq$ `int`, on a `int ->float` $\subseteq$ `float->float`

Typage

Typage

Donner un type à une expression c'est indiquer la nature d'une expression.

- Objectifs:
  - Vérifier les appels de fonctions.
  - Valider le code (homogénéité).
  - Gérer la mémoire.
- Typage plus ou moins forts
  - OCaml: `float_of_int(x) +.  2.3`
  - Javascript: `(2 + 3) + " saucisses"`
- Typage explicite: le programmeur doit lui-même indiquer les types (déclarations).
- Typage implicite: le type est inféré par un programme (algorithme d'unification).

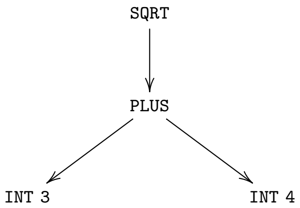
Sous-Typage

Définition

Un type A est un sous-type de B si toutes les expressions (les objets) de type A sont aussi de type B.

-
  - `int` est un sous-type de `float`.
  - "entier naturel" est un sous-type de "entier".
  - "poisson" est un sous-type de "animal".
- Si on a besoin d'une expression de type B, et que A est un sous-type de B, on peut prendre une expression de type A.
  - si f prend un entier, je peux calculer $f(3)$.
  - si j'ai besoin d'un animal, je peux prendre un poisson.
- Attention au sens:
  - si f prend un entier naturel, je ne peux pas (forcément) calculer $f(-3)$.
  - si j'ai besoin d'un poisson, je ne peux pas (forcément) prendre un serpent.
- Dans les signatures des fonctions: + général pour les paramètres, + particulier pour le résultat.
- Héritage dans les langages objets (11).

Grammaires d'expressions

- Compilation: domaine de l'informatique qui s'intéresse à la traduction d'un langage dans un autre.
- Fondement de la programmation: traduction d'un langage "compréhensible" (Python) en langage machine.
  - point de détail: Python est interprété et non compilé.
- Analyse lexicale: séparation du code en jetons.
  - `math.sqrt(3 + 4)` $\rightarrow$ reconnaitre `sqrt`, 3, 4, l'opérateur `+`, les parenthèses.
- Analyse syntaxique: organisation des jetons en arbre syntaxique.
  - 

Grammaires d'expressions (II)

- Les Grammaires permettent d'exprimer le code reconnaissable par le compilateur/l'interprêteur.
- Définition à l'aide de "graines" `S ::= E1 | E2 | ... | EN`
  - formellement, point fixe d'une fonction (théorème de Knaster-Tarski).
- Grammaire des entiers `N ::= 0 | Succ(N)`
- Grammaire de l'arithmétique `N ::= 0 | Succ(N) | Plus(N,N) | Sous(N,N) | Mult(N,N)`
- Grammaire de la Carte de référence.

Effets de bords

Définition

Un effet de bord est une instruction d'une fonction qui modifie un état (la mémoire, l'affichage) autre que la valeur de retour de la fonction.

- souvent son interprétation n'a pas d'effet direct sur le calcul.

- Affichage: `print` est un effet de bord, elle affiche sur la sortie standard.
- la modification de fichiers ("disque dur") est un effet de bord.
- Nécessaires, mais difficile à analyser.
  - idempotence des fonctions ?
- `print` fait un effet de bord: affichage à l'écran
  - utile pour connaître les valeurs intermédiaires des variables.

Valeurs Intermédiaires

```
def essai_var3(x : int) -> int:

    n : int = 0
    print("la valeur de n est:", format(n))

    m : int = x
    print("la valeur de n est:", format(n), "la valeur de m est:", format(m))

    n = m + x
    print("la valeur de n est:", format(n), "la valeur de m est:", format(m))
    m = n + 1
    print("la valeur de n est:", format(n), "la valeur de m est:", format(m))
    n = m + x
    print("la valeur de n est:", format(n), "la valeur de m est:", format(m))
    return n
```

- A utiliser en TME.

Fonctions partielles

- primitive **`print`**:
  - utilisation courante: afficher des chaînes de caractères.
  - peut contenir des expressions de différents types.
  - la valeur de retour de **`print`** est

Fonctions partielles

- primitive `print`:
  - utilisation courante: afficher des chaînes de caractères.
  - peut contenir des expressions de différents types.
  - la valeur de retour de `print` est Rien

Fonctions partielles

- primitive `print`:
  - utilisation courante: afficher des chaînes de caractères.
  - peut contenir des expressions de différents types.
  - la valeur de retour de `print` est Rien (en `Python`: `None`).
  - le type de `None` est

Fonctions partielles

- primitive **print**:
  - utilisation courante: afficher des chaînes de caractères.
  - peut contenir des expressions de différents types.
  - la valeur de retour de **print** est Rien (en Python: None).
  - le type de None est None:
- Fonctions qui n'ont pas de valeur de retour:

```
def affiche_trois_fois(n : int) -> None:
    print(n)
    print(n)
    print(n)

assert affiche_trois_fois(10) == None
```

  - Est-ce vraiment des fonctions ?
  - Plus tard dans l'UE, types optionnels
    - renvoyer soit un entier (quand ça "marche"), soit rien (quand ce n'est pas possible)

Différence entre `print` et `return`

- `print` est une instruction qui affiche la valeur d'une expression sur la sortie standard.
- `return` renvoie la valeur de son argument à l'appelant.
  - Si l'appelant est le *top-level* de `mrpython`, il affiche la valeur qu'il reçoit.
  - Si l'appelant est une expression, il utilise cette valeur.

```
def h2(x : int) -> int:
    return x + 1

def h3(x : int) -> None:
    print(x + 1)
```

Comparer les expressions `1 + 2 * h2(10)` et `1 + 2 * h3(10)`

Variables Globales

- On peut affecter des variables en dehors des fonctions ("globales").
  - elle doivent être déclarées.
- Ces variables ne sont pas accessibles dans les fonctions.
- Ces variables ne sont pas modifiables.
- Ces variables sont, en fait, des constantes.
- Utiles pour les tests et les essais.
  - surtout avec des structures de données (cours 05-10).

```
nombre : int = 42

def increm(x : int) -> int:
    return x + 1

assert increm(nombre) == 43

def ajoute_n(x : int) -> int:
    return x + nombre # ERREUR

nombre = nombre + 1 # ERREUR
```

Modèle mémoire

- Mémoire est un espace indicé:
  - chaque "tiroir" a une taille et une adresse.
- une variable, c'est un nom pour l'adresse d'un tiroir,
  - une table de symboles lie noms et adresses.
- deux "zones" de mémoire:
  - le tas: où vivent les variables globales, les données, les objets, les fonctions (le code),
  - la pile: qui sert à l'execution de fonction.
    - contient les variables locales et les arguments,
    - durée de vie limitée,
    - cas des fonctions qui appellent d'autres fonctions
- En LU1IN002: modèle mémoire formel.

Décidabilité

Définition

Un problème de décision est décidable quand il existe un algorithme pour le résoudre.

- Problème de décision: résultat booléen.
- Solution: un unique algorithme qui marche dans tous les cas particuliers.
- Primalité d'un entier.
  - décider si un entier est premier $\texttt{div}(n) = \{1, n\}$
  - entrée: un entier, résultat: booléen.
  - algorithme: crible d'Eratosthene (par exemple)

Décidabilité informatique vs. Décidabilité logique

- Décidabilité logique: une formule est décidable (par rapport a un système logique) quand il existe une preuve (dans le système logique) de sa vérité ou de sa fausseté.
- En fait c'est pareil (Curry-Howard).

Indécidabilité

- Il existe des problèmes indécidables.
  - des problèmes qu'aucun algorithme ne peut résoudre.

Indécidabilité

- Il existe des problèmes indécidables.
  - des problèmes qu'aucun algorithme ne peut résoudre.

Théorème: Incomplétude de Gödel

Tout système logique un peu intéressant contient au moins une formule indécidable.

- "un peu intéressant": contient l'arithmétique de Peano (0, $\mathtt{S}$, $+$, .)
- Logique $\rightarrow$ Informatique: il existe des problèmes indécidables dès que l'expressivité est suffisante.
- Utilisée (de manière discutable) en philosophie (cf. Debray, Bouveresse)
- Exemple: Correspondance de Post
  - Instance: dominos, chacun en quantité illimitée: $\left(\frac{a}{baa}\right)\left(\frac{ab}{aa}\right)\left(\frac{bba}{bb}\right)$
  - Question: existe t-il une suite (finie) de dominos telle que le mot lu au-dessus est le même que le mot lu en dessous ?

Indécidabilité

- Il existe des problèmes indécidables.
  - des problèmes qu'aucun algorithme ne peut résoudre.

Théorème: Incomplétude de Gödel

Tout système logique un peu intéressant contient au moins une formule indécidable.

- "un peu intéressant": contient l'arithmétique de Peano $(0, \mathtt{S}, +, .)$
- Logique $\rightarrow$ Informatique: il existe des problèmes indécidables dès que l'expressivité est suffisante.
- Utilisée (de manière discutable) en philosophie (cf. Debray, Bouveresse)
- Exemple: Correspondance de Post
  - Instance: dominos, chacun en quantité illimitée: $\left(\frac{a}{baa}\right)\left(\frac{ab}{aa}\right)\left(\frac{bba}{bb}\right)$
  - Question: existe t-il une suite (finie) de dominos telle que le mot lu au-dessus est le même que le mot lu en dessous ?
  - Ici: $\left(\frac{bba}{bb}\right)\left(\frac{ab}{aa}\right)\left(\frac{bba}{bb}\right)\left(\frac{a}{baa}\right)$, mot *bbaabbbaa*
  - Il n'existe pas d'algorithme qui prend en entrée un jeu de domino et décide la question.
    - PCP est indécidable.

Terminaison

- Une boucle s'arrête quand sa condition est fausse.
  - peut-on être sur qu'elle sera forcément fausse au bout d'un certain temps?

```python
def infini() -> int:
    """compte pendant l'eternite et renvoie 1 ensuite  """

    i : int = 0 # compteur

    while True:
        i = i + 1
    return 1
```

-

```python
def somme_entiers2(n : int) -> int:
    """retourne la somme des n premiers entiers naturels.  """
    i : int = 1 # entier courant, en commencant par 1
    s : int = 0 # la somme cumulee
    while i <= n:
        s = s + i
        i = i - 1
    return s
```

- Peut-on détecter les programmes divergents ?
- La Terminaison est t-elle décidable ?

Problème de l'arrêt

- Supposons qu'on a la fonction (non-typée) suivante:

```python
def arret(fonc, argu ):
    """renvoie True si l'appel de fonction fonc avec l'argument argu termine, False sinon."""
```

- On définit alors:

```python
def diago(f):
  i = 0
  if arret(f,f):
      while True:
          i = i + 1
  else:
      return i
```

- Que dire de `diago(diago)` ?

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Problème de l'arrêt

- Supposons qu'on a la fonction (non-typée) suivante:

```
def arret(fonc, argu ):
    """renvoie True si l'appel de fonction fonc avec l'argument argu termine, False sinon."""
```

- On définit alors:

```
def diago(f):
    i = 0
    if arret(f,f):
        while True:
            i = i + 1
    else:
        return i
```

- Que dire de `diago(diago)` ?
  - si `diago(diago)` s'arrête, c'est que `arret(diago,diago)` vaut `False`. Contradiction !
  - si `diago(diago)` ne s'arrête pas, c'est que `arret(diago,diago)` vaut `True`. Contradiction !
  - On a montré par l'absurde, qu'il n'existe pas de fonction `arret`.
- La terminaison d'un programme est indécidable.
  - énormes conséquences pour l'informatique.

Conclusion

- boucle: instruction `while`.
- simulation de boucles.
- boucles imbriquées.

Conclusion (II)

TD-TME 02

- Thèmes 02 et 03 du cahier d'exercices.

Activité 02

- Approximation de pi

Cours 03 - 26/09/2022

- "ces boucles qui nous gouvernent" (deuxième partie)