Éléments de Programmation + Cours 1 - Fonctions, Variables, Alternatives

Romain Demangeon

LU1IN011 - Section ScFo 11 + 13

12/09/2022

Enseignant

- Romain Demangeon, maître de Conférences au LIP6, équipe APR
- romain.demangeon@sorbonne-universite.fr
- Bureau 25-26.314 - 0144278825 (0626641354)
- Transparents de cours:
  - Moodle LU1IN011.

Étudiants

- ScFo : LU1IN011 obligatoire, pas de NSI au lycée.
  - LU1IN021 : ScFo NSI.
  - LU1IN001 : pas ScFo (6ECTS)
- Première année.

Prérequis et Organisation hebdomadaire

Prérequis

- Pour les Cours: aucun prérequis en programmation.
- Pour les Exemples: Mathématiques de Terminale.

- 1h45 de Cours en amphithéâtre.
- 1h45 de Travaux Dirigés: exercices corrigés sur feuille et au tableau.
- 1h45 de Travaux sur Machines Encadrés: exercices pratiques sur ordinateur.
- 1h45 d'Activités: TMEs plus libres et plus appliqués.
- travail en autonomie: apprentissage du cours, entraînement aux exercices de TDs, entraînement à la programmation pratique.

TDs, TMEs et Activités commencent le Vendredid 16 septembre.

Environnement

- IDE fait-maison: *MrPython*.
  - présent dans les salles de TMEs (libre-service),
  - téléchargeable pour le travail en autonomie
    - installeur Windows 10.
    - procédure Linux.
- Proche de IDLE
- Utilise *Python 101*, le langage du cours,
  - "Si *MrPython* n'accepte pas l'expression, les correcteurs d'examens non plus".
- Force les annotation de types.

Evaluation

- Note sur 100:
  - Contrôle Final en janvier 50.
  - Devoir sur Table le 18/11 à 18h30 à Jussieu 15,
  - TME solo fin novembre 15,
  - Evaluations TD (interro + colles/devoirs/participation) 5,
  - Rendus de TME systématiques 5.
  - Rendus d'Activités systématiques 10.
- Première session validée si la note totale ≥ 50.
- Deuxième chance, contrôle écrit en juin, note sur 100.
  - on prend la meilleure des deux notes.

Informations pratiques

- Groupes de TD:
  - cf. tableau.
- Consignes:
  - Se munir de ses identifiants (login/mot de passe donnés avec la carte d'étudiant) pour les TMEs.
  - Ne pas changer de groupe de TD.
  - Transmettre les justificatifs d'absence à Patricia Lavanchy (24.25.204) `Patricia.Lavanchy@ufr-info-p6.jussieu.fr`

Ressources

- Moodle du cours LU1IN011.
  - Première source pour l'UE.
- Livre du cours.
  - *Eléments de Programmation, de l'algorithme au programme Python* Peschanski-[D], édition Ellipses
    - bibliothèques de Jussieu, librairies, vente en ligne.
  - `.pdf` (support de cours) disponible légalement et gratuitement sur le site de l'UE.
- Cahiers d'exercices:
  - deux parties.
  - certains exercices identifiés comme corrigés.
  - corrections en fin de cahier.
- Carte de références:
  - seul document autorisé aux examens (amulette).
- Transparents de cours:
  - Mis sur Moodle chaque semaine.
  - Vidéos de cours (+ premiers TDs) de LU1IN001-2020.
- *MrPython*:
  - Téléchargeable depuis le site de l'UE (installeur Windows 10).

Supports

- Tous les supports sont distribués aux étudiants par l'Association Ludique et InformAtique de Sorbonne université (ALIAS).

- Première distribution à partir du 20/09 :
  - Cahiers d'exercices TD/TME (2021-2022).
    - première partie.
    - version étudiant.
  - Carte de référence
- Pas nécessaire pour le 1er TD-TME.
  - les sujets sont sur Moodle.
- Les sujets d'activité ne sont pas imprimés.
  - consulter directement sur Moodle.

Correspondance TD/Thème

A chaque semaine de TD-TME, des thèmes du cahier d'exercices correspondent :

- TD 1 Thème 01-02
- TD 2 Thème 02-03
- TD 3 Thème 03-04
- TD 4 Thème 05
- TD 5 Thème 06
- TD 6 Thème 07-08
- TD 7 Thème 11
- TD 8 Thème 09
- TD 9 Thème 09-10
- TD 10 Thème 12
- TD 11 Révisions

Conseils pour Réussir l'UE

- Ne pas se sous-estimer: pas de pré-requis informatiques pour ce cours, accessible à tout étudiant de L1.
- Ne pas se sur-estimer: des connaissances en informatique (lycée/autodidaxie) et en Python ne garantissent pas la réussite de l'UE. Vision scientifique de l'informatique et de la programmation.
- Aller en cours pour prendre des notes et participer: la première visualisation des concepts est importante, l'écriture des programmes permet de s'entraîner.
- Etre actif en TD/TME/Activité: attention aux binômes "déséquilibrés" et à l'attitude passive en TD.
- Travailler seul: refaire les exercices de TD/TME si nécessaire, faire ceux non-traités en TD/TME. Proposer une démarche personnelle pour les activités.
- Parler avec l'équipe pédagogique: poser des questions pendant (ou à la fin) des amphis, en TD/TME, envoyer des mails à l'équipe pédagogiques pour des précisions, des corrections, des énoncés supplémentaires, des exemples d'interrogations ou d'examens.
- Ne pas compter sur la seconde chance.

LU1IN001 / LU1IN011

- Jusqu'en 2020 : une unique UE d'informatique au L1S1 pour tous, LU1IN001 - 6ECTS (50 groupes).
- Maintenant, 4 UEs :
  - LU1IN001 - 6ECTS : public non-ScFo (22 groupes).
  - LU1IN011 - 9ECTS : public ScFo non-NSI (15 groupes).
  - LU1IN021 - 9ECTS : public ScFo NSI (2 groupes).
  - LU1INMA1 - 9ECTS : option ScFo (10 groupes).
- 011 est une version renforcée de 001;
  - programme légèrement plus rapide.
  - contenu supplémentaire : récursion et tableaux.
  - 1h45 d'activités en plus par semaine.
- intersection conséquente:
  - même livre,
  - même cahiers d'exercices,
  - examens communs (à un exercice près).

Activités

- En salle machine :
  - TME : 2 ou 3 exercices typiques, semblables aux exercices de TDs, décomposés en questions.
  - Activités : mini-projets hebdomadaires en 1h45, grande liberté dans les solutions possibles, structure personnelle, applications plus concrètes.
- Exemples : images de fractales, traitement des données, génération de labyrinthes.

Cadre: Informatique et Programmation

Informatique

- Dualité technique/science.
- LU1IN0*1: Importance de l'approche scientifique (vs. "bricolage").

- Science de l'information et du calcul.
- Englobe (entre autres):
  - Mathématiques discrètes: algorithmique, modèles de calculs, logique, complexité, concurrence, ...
  - Programmation: langages, typage, sémantique, compilation, ...
  - Architecture: multi-processeurs, puces, calcul flottants, ...
  - Réseaux, Robotique, I.A., ...
- Programmer (écrire des programmes), c'est donner des instructions à une machine en vue de lui faire réaliser un résultat.
  - En LU1IN0*1: résolution de problèmes.

Python

"LU1IN0*1 sont des cours de programmation en Python, et non des cours de Python."

- Écrire du "vrai" Python 3.
- Sans utiliser toutes les fonctionnalités de Python (*Python 101*).
  - *MrPython* est moins permissif que l'interpréteur Python.
  - *Python 101* est le langage des examens !
- Objectif principal:
  - Comprendre des éléments de programmation généraux (et transposables).
- Objectifs secondaires:
  - Connaître des bases d'un langage en particulier.
  - Etudier un axe de la programmation: la sûreté.

LU1IN0*1 - Éléments de programmation (+)

Concepts fondamentaux

- Résolution de problèmes (vision de l'ingénieur).
  - "un probleme, une fonction"
- Démarche:
  - Spécification de problèmes informatiques,
  - Approche algorithmique des solutions,
  - Programmation dans le langage *Python 101*.

Éléments abordés

- Expressions. (01)
- Fonctions. (01)
- Alternatives. (01)
- Boucles. (02, 03)
- Structures de données: listes (05,06,07,08), n-uplets (07), chaînes (04), ensembles (09,10), dictionnaires (09,10), objets (11).
- Récursion (07)

Une Fonction

```
def liste_premiers(n : int) -> List[int]:
    """Precondition : n >= 0

       Retourne la liste des nombres premiers inferieurs a n."""

    i_est_premier : bool = False  # indicateur de primalite

    L : List[int] = []   # liste des nombres premiers en resultat

    i : int # parcourt les entiers
    for i in range(2, n):
        i_est_premier = True

        j : int # parcourt les diviseurs possibles
        for j in range(2, i - 1):
            if i % j == 0:
                # i divisible par j, donc i n'est pas premier
                i_est_premier = False

        if i_est_premier:
            L.append(i)

    return L
```

Une Fonction

```
def liste_premiers(n : int) -> List[int]:
    """Precondition : n >= 0

       Retourne la liste des nombres premiers inferieurs a n."""

    i_est_premier : bool = False  # indicateur de primalite

    L : List[int] = []   # liste des nombres premiers en resultat

    i : int # parcourt les entiers
    for i in range(2, n):
        i_est_premier = True

        j : int # parcourt les diviseurs possibles
        for j in range(2, i - 1):
            if i % j == 0:
                # i divisible par j, donc i n'est pas premier
                i_est_premier = False

        if i_est_premier:
            L.append(i)

    return L
```

- En-tête: `def liste_premiers(n : int) ->List[int]:`
  - paramètres `n : int`, type de retour `-> List[int]`
- Variables: `i_est_premier = False`
- Expressions: `1`, `i - 1`, `i % j == 0`
- Applications: `liste_premiers(10000)`
- Instructions: alternative `if i_est_premier:` ... (Cours 2)
  boucle `for j in range(2, i - 1):` ... (Cours 3)
- Commentaires, Spécification, Indentations.

Programmer

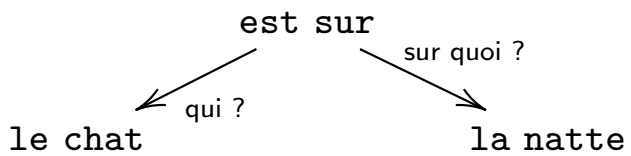
- Programmer c'est donner des ordres à une machine.
  - pour calculer des valeurs,
  - pour modifier une états (écran, fichier),
- L'interface entre la machine et le programmeur est le langage de programmation.
  - comme un langage courant (le français), un langage de programmation est un ensemble de règles qui donne du sens à des textes.
- Les ordres donnés à la machine sont écrits, par le programmeur, comme un texte dans le langage de programmation.
- Un programme qui s'exécute sur la machine, l'interpréteur (par exemple *MrPython*) ou le compilateur:
  - comprend le texte,
  - executent les ordres demandés par le texte.
- Une manière simple de programmer est de demander à la machine d'évaluer des expressions
  - par exemple calculer que `208/2*(2+2)` vaut `416.0`

Expressions

Définition

- [Livre] Une expression est une écriture textuelle bien formée que l'on peut saisir au clavier.
- [Formellement] Une expression est un élément d'une algèbre de termes, définie inductivement à partir d'une signature.
- [Sérieusement] Une expression c'est un ensemble de mots (jetons) qui sont arrangés les uns avec les autres.

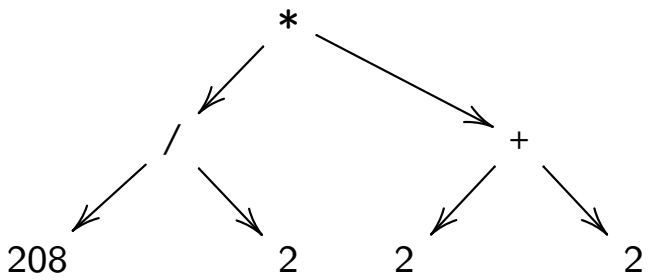
Expressions des langages courants

- Comment l'interprèteur comprend ce qu'on écrit ?
- Analogie avec les langages courants:
  - 1. une suite de symboles: `le chat est sur la natte`
  - 2. un découpage en jetons: `le`, `chat`, `est`, `sur`, `la`, `natte`.
  - 3. un arrangement des jetons les uns par rapport aux autres.
    - 

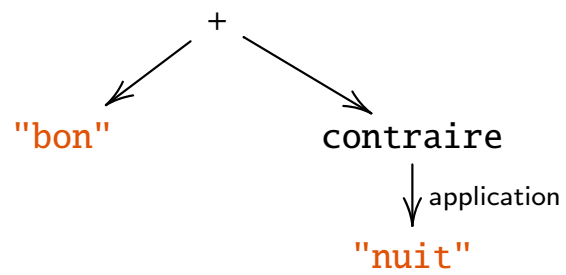
    - `être sur` demande un sujet et un complément, et donne une phrase.
    - `le/la` précise le nom qui suit
- Ces étapes s'appellent l'analyse lexicale et syntaxique.
- Elle construit du sens (une structure) à partir d'une suite de symbole.
- Elle rejette les écritures qui n'ont pas de sens:
  - `chat natte le la` ou `2/+3.4.5`
- Les analyses lexicales et syntaxiques sont étudiées en Licence d'Informatique (mais pas en LU1IN0*1).

Expressions Informatiques

- Il se passe la même chose pour l'interpréteur *Python*
- 1. suite de symboles `208 / 2 * (2 + 2)`
- 2. découpage: 208, 2s, division, addition, multiplication, parenthèse
- 3. arrangement: régles de priorité et parenthèsage



- 1. `"bon"+ contraire("nuit")`
- 2. `"bon"`, `"contraire"`, addition, application, fonction `contraire`.
- 3.



Evaluation

Définition

Evaluer une expression c'est calculer sa valeur.

- Première utilisation d'une machine (Pascaline, TI-92+, *MrPython*):
  - évaluer des expressions.
- Exemple:
  - `208/2*(2+2)` s'évalue en

Evaluation

Définition

Evaluer une expression c'est calculer sa valeur.

- Première utilisation d'une machine (Pascaline, TI-92+, *MrPython*):
  - évaluer des expressions.
- Exemple:
  - `208/2*(2+2)` s'évalue en `416.0`
  - `"bon"+contraire("nuit")` s'évalue en `"bonjour"`.
    - si on a définit correctement la fonction `contraire`.
- On distingue deux sortes d'expressions:
  - Expressions atomiques (ou simples):
    - objets informatiques manipulables les plus simples (pas besoin d'arranger des jetons),
    - nombres, chaînes,
    - s'évaluent en elles-même.
  - Expressions composées:
    - formées de sous-expressions (simples ou composées) arrangées entre elles
    - expressions arithmétiques: sommes, produits, applications de fonctions (max), ...
    - on calcule a valeur de manière inductive (en calculant d'abord la valeur des sous-expressions).

Expressions atomiques

- Une expression atomique `v` s'évalue en `v`, sans calcul.

```
=== Evaluation de : '42' ===
42
============================
```

- L'évaluation est un processus complexe:
  1. Lecture de l'expression (analyse lexicale et syntaxique).
  2. Représentation interne de l'expression (codage en objet).
  3. Evaluation de l'expression: résultat (ou non).
  4. Affichage du résultat.
  - un seul `42` "à l'oeil nu", mais plusieurs entités informatiques.
- Une expression possède un type qu'on peut obtenir avec `type`.
  - Un type renseigne sur la nature d'une entité informatique.
  - La représentation interne d'un type est une classe d'objets.
  - L'étude et l'utilisation des types sont au programme de LU1IN0*1.

Types d'expressions atomiques

On décrit les expressions atomiques par types:

Booléens

Expressions logiques, indispensables pour calculer (alternative, boucles).
`True` signifie vrai et `False` signifie faux.
Leur type est `bool`.

Entiers

Écrits en notation décimale usuelle, de taille arbitraire.
Leur type est `int`.

Types d'expressions atomiques (suite)

Constantes à virgule flottante

Nombres avec virgule (`3.14`) et/ou exposant (`-4.3e-3`).
Approximations informatiques de nombres rationnels / réels.
Branche de l'informatique à part entière
  représenter $\mathbb{Q}$ ou $\mathbb{R}$ dans une machine finie
  Python tout nu est très mauvais pour les manipuler (`0.1 + 0.2`).
Leur type est `float`.
Les entiers sont convertis en flottants si nécessaire `int` $\subsetneq$ `float`.

Chaînes de caractères

Texte encadré par des apostrophes ou guillemets doubles (`"bonjour"`).
Objets du Cours 04.
Leur type est `str`.

Expressions composées: Opérateurs Arithmétiques

Définition

- Les expressions composées sont formées de combinaisons de sous-expressions, atomiques ou elles-mêmes composées.
- La combinaison de sous-expressions se fait avec des fonctions soit primitives, soit définies par l'utilisateur.

Premier cours/TD/TME:

- Expressions arithmétiques.
- Primitives: opérateurs numériques.
- Fonctions définies par l'utilisateur: calculs de formules.

Primitives numériques:

- opérateurs binaires (et unaire) sur les nombres: addition, soustraction, produit, divisions, l'opposé, ...
- Les parenthèses permettent de construire correctement un terme.
  - `208/2*(2+2)` vs. `208/(2*(2+2))`

Expressions composées: Opérateurs Arithmétiques (suite)

Divisions

Deux types de divisions:

- Division entière (ou division euclidienne) `//` : prend deux entiers, rend un entier.
- Division flottante `/`: prend deux flottants (ou entiers qui sont convertis), rend un flottant.

Le reste de la division euclidienne s'obtient avec le modulo `%`.

- En l'absence de parenthèses, la priorité des opérateurs est standard (comme en mathématiques).

Expressions composées: Application de Primitives

- Python dispose de plusieurs fonctions prédéfinies (primitives), accessibles plus ou moins directement.
- Pour appliquer une fonction à une expression, il faut connaître sa spécification, incluant, entre autres, les types de ses entrées et de son résultat et les préconditions d'application.
  - en mathématiques on écrit par exemple $f : \mathcal{L}(\mathbb{R}^n, \mathbb{R}^m) \to \mathbb{N}$
- En Python, la fonction `math.sqrt` par exemple est spécifiée par:

```
def math.sqrt(x : float) -> float:
  """Precondtion: x >= 0
     retourne la racine carree de x."""
```

- La spécification d'une fonction se compose de:
  1. une en-tête qui donne le nom, ses paramètres, le type des paramètres, et le type du résultat
  2. (si nécessaire) des préconditions pour application de la fonction.
  3. une description qui indique quel problème résout la fonction.

Expressions composées: Applications de Primitives (II)

- Pour utiliser une primitive (en fait, n'importe quelle fonction) on écrit une application:
  - la syntaxe est:
    `<nom de la fonction>(<expression1>, <expression2>, ...)`
  - par exemple `math.sqrt(2)`
  - ou `contraire("bonjour")`
  - ou `max(1 + 1 , 3 * 1 , 2 + (2 * 1))`
- Une application est une expression composée
- Pour l'évaluer il faut que la fonction existe (que l'interpréteur la connaisse déjà)

Expressions composées: Applications de Primitives (III)

Principe d'évaluation

Pour évaluer `fun( arg1, arg2, ..., argn)`:

1. On commence par évaluer les arguments `arg1`, puis `arg2`, `...`, puis finalement `argn` (dans cet ordre).
2. On calcule le résultat de la fonction (on effectue son corps) en remplaçant les paramètres par les arguments.
3. On renvoie la valeur de retour de la fonction à l'appelant:
   - si la fonction est appelée dans la zone d'évaluation, on affiche la valeur,
   - si la fonction est appelée dans une expression, on remplace l'appel par sa valeur.

- Les primitives autorisées en LU1IN0*1 sont listées sur la carte de référence.
- les primitives sont classées par Python en bibliothèques appelées modules.
  - pour charger un module on utilise `import`
  - exemple de module: `math` qui contient, entre autres, `math.sin` et `math.pi`.

Evaluation des expressions arithmétiques

Principe général

Pour évaluer une expression composée `e`:

1. Si `e` est atomique, on retourne la valeur `e`.
2. Si `e` est composée, alors:
   - 2.1 On détermine la sous-expression `e1` de `e` à évaluer en premier.
     - On procède de gauche à droite et de bas en haut (dans "l'arbre").
   - 2.2 On évalue `e1`.
   - 2.3 On évalue `e` dans laquelle on remplace `e1` par sa valeur.

Exemple: Evaluation de `(3 + 5) * ( max(2, 9) - 5 ) - 9`

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:

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

-

```
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.

```
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.

```
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

```
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`.

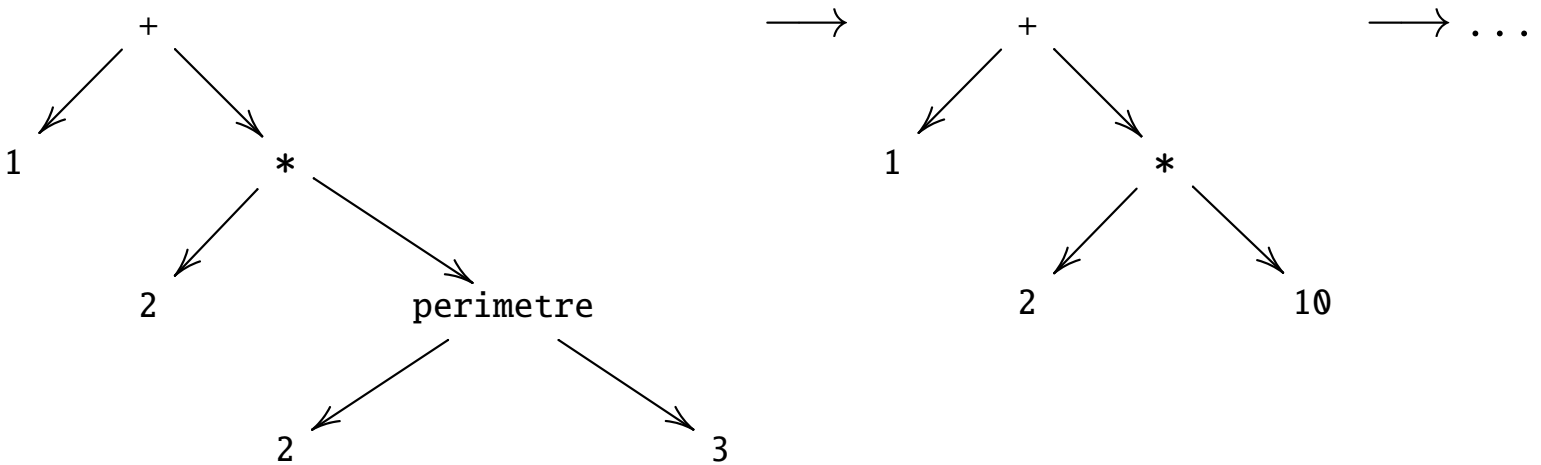
Valeur de retour

- 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
==================================
```

Variables et Affectations

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

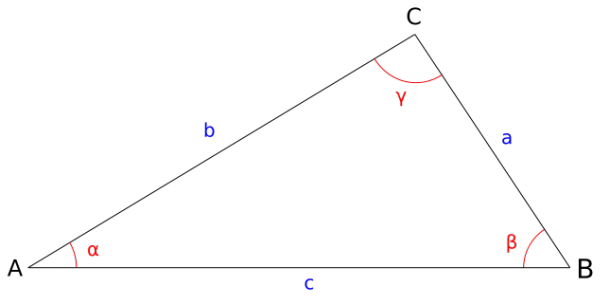
```
def perimetre(largeur : int, longueur : int) -> int:
    """ precondition : (longueur >= 0) and (largeur >= 0)
    precondition : longueur >= largeur
    retourne le perimetre du rectangle defini par sa largeur et sa longueur."""

    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.

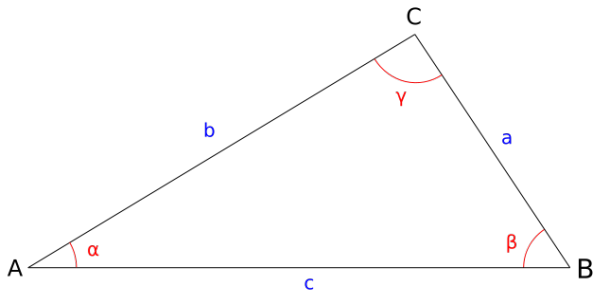
Problème

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 préconditions
   - 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 préconditions
   - ici: les longueurs sont positives,
   - ici: elles correspondent à un triangle (par exemple, pas `3`, `4`, `10`).

```
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.

```python
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

```python
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, $\bot$).
- 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

```
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.

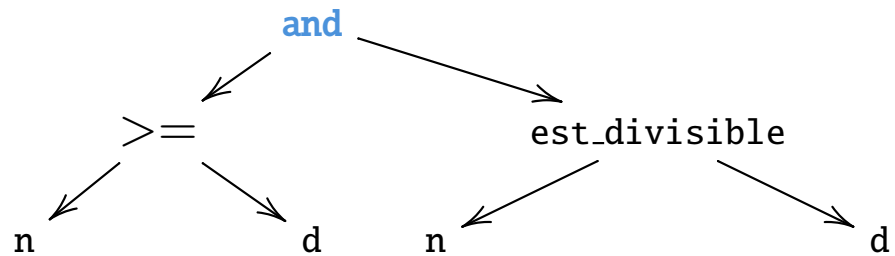
- une application d'un prédicat à des arguments (e.g. `est_divisible(12,4)`) est une expression booléenne (composée).
- on peut composer les opérateurs logiques et les prédicats pour obtenir des expressions booléennes complexes.
  - exemple: `(n >= d) and est_divisible (n, d)`

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

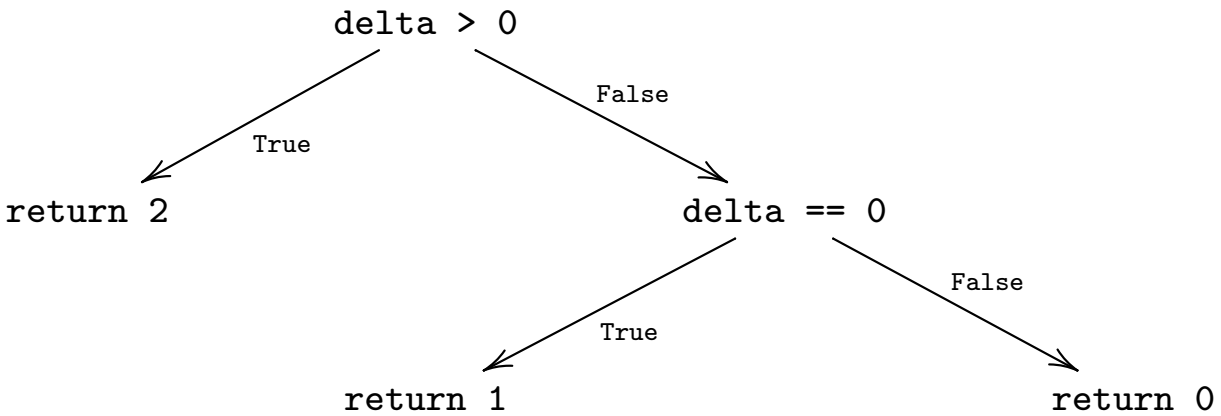
Problème

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 $a.x^2 + b.x + c = 0$.

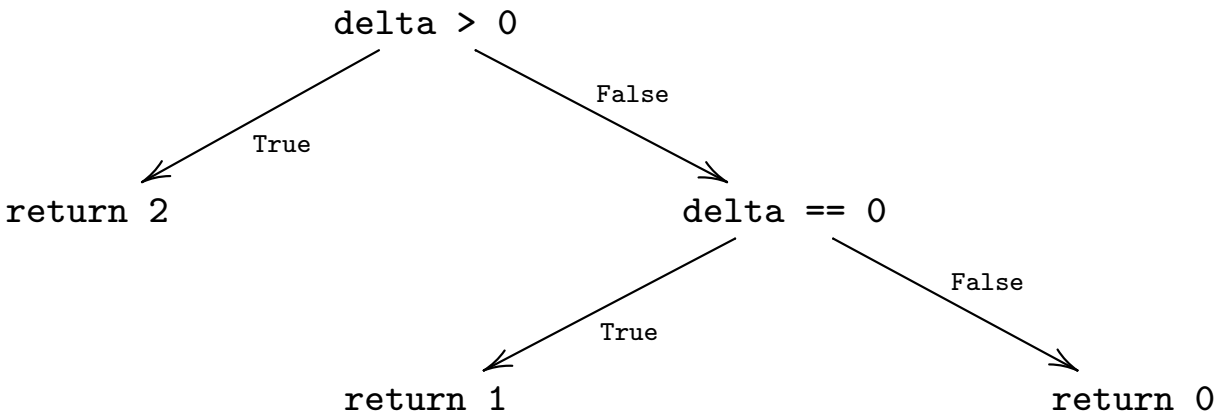
- On sait qu'il faut calculer le discriminant et discuter:
  - si le discriminant est strictement positif il y a 2 solutions,
  - si le discriminant est nul il y a 1 solution,
  - si le discriminant est négatif il n'y a aucune solution.
- On a un choix en trois cas.
- On imbrique deux alternatives:

```
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:

```
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.

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

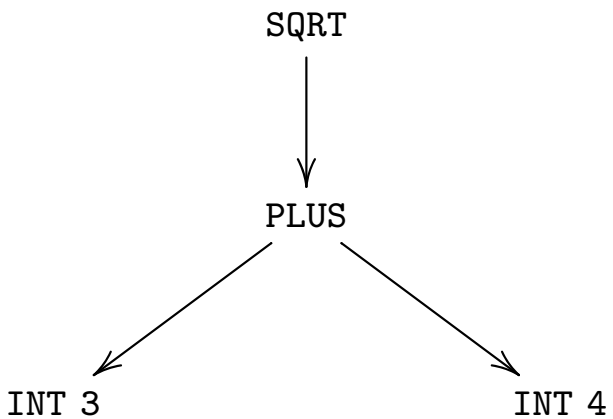
Définition

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)` $\longrightarrow$ 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 ::= O | Succ(N)`
- Grammaire de l'arithmétique `N ::= O | 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.

Conclusion

- Programmation par fonctions:
  - problème générique : énoncé.
  - solution générique : fonction.
- Sûreté :
  - spécification systématique,
  - annotations de types,
  - jeu de tests.
- Briques de base :
  - expressions (inclue appels de fonctions).
  - définition de fonctions.
  - séquence, définition et affectation de variables.
  - alternatives.

Conclusion (II)

TD-TME 01

- Thèmes 01 et 02 du cahier d'exercices.

Activité 01

- Cadavre exquis (génération de textes aléatoires / contrôlée).

Cours 02

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