

Structures de base [bs]

Support de Cours

Karine Zampieri, Stéphane Rivière

Unisciel  algoprog  Version 12 mai 2018

Table des matières

1	Le langage	3
1.1	Le langage	3
1.2	Les mots-clés	4
1.3	Les identifiants	5
1.4	Règles d'écriture d'un identifiant	6
1.5	Séparateurs et espaces blancs	7
1.6	Primitives	8
2	Variables, types et valeurs	9
2.1	Variable, type et valeur	9
2.2	Types intégrés	10
2.3	Littéraux	11
2.4	Exemples : Variables, types et valeurs	12
3	Déclarations	13
3.1	Déclaration de variables	13
3.2	Définition de constante	14
3.3	Exemples : Déclarations	15
4	Structure générale	16
4.1	Commentaire	16
4.2	Instruction composée	17
4.3	Structure générale	19
4.4	Exemple : Le plus petit algorithme	20
5	Interactions avec l'extérieur	21
5.1	Interactions avec l'extérieur	21
5.2	Saisie de données	22
5.3	Affichage de résultats	23
5.4	Exemples : Communication des données/résultats	24
6	Expressions arithmétiques	25

6.1	Qu'est-ce qu'une expression ?	25
6.2	Opérateurs arithmétiques	26
6.3	Évaluation des expressions	27
6.4	Exemples : Opérateurs arithmétiques	28
6.5	Cohérence de type numérique	29
6.6	Fonctions mathématiques	30
7	Affectation interne	31
7.1	Affectation interne	31
7.2	Exemples : Affectation interne	32
7.3	Exemple : Affectations et conversions	33
8	Trace d'exécution d'un script	34
8.1	Trace d'exécution	34
8.2	Exemple complet de programme / pgdiffsecs	36
8.3	Erreur à l'exécution	38
8.4	Méthode	39
9	Compléments	40
9.1	Variables et constantes	40
9.2	Affectation : Compléments	42
9.3	Transtypage	43

Python - Structures de base (Cours)



Mots-Clés Structures de base, Atomes du langage, Déclarations, Structure d'un algorithme, Interactions avec l'extérieur, Expressions algébriques, Affectation interne ■

Optionnel L'Algorithmique ■

Difficulté ● ○ ○



Introduction

Ce module traite des **structures de base** : atomes du langage, variables, types et valeurs, déclarations, structure d'un script, interactions avec l'extérieur, expressions algébriques, affectation interne, puis il détaille la trace d'exécution d'un script. Le *Compléments* donne quelques précisions sur les variables, l'affectation et le transtypage.



Remarque

Pour certains, ce module apparaîtra comme une suite d'évidences, pour d'autres comme un foisonnement de choses nouvelles aux définitions contraignantes. Que ces derniers prennent le temps de faire les exercices. L'essentiel en algorithmique/programmation est dans la pratique.

1 Le langage

1.1 Le langage



Règles, Alphabet du langage, Éléments lexicaux

Tout **langage** de programmation possède :

- Des **règles de présentation** (aspect lexical).
- Des **règles d'écriture** (une syntaxe).
- Des **règles d'interprétation** (une sémantique).

L'**alphabet du langage** permet de construire les mots et les expressions. Comme toute phrase d'un langage courant, il est composé de mots appelés **lexèmes** ou **éléments lexicaux** (*tokens*).



Le langage Python

Langage de scripts *orienté objet*.



Alphabet du langage

C'est celui de la langue anglaise (majuscules et minuscules), les chiffres 0-9, quelques symboles spéciaux (ceux des mathématiques par exemple) et quelques symboles de ponctuation. Le codage étant en UTF8, les lettres accentuées sont autorisées dans les lexèmes.



La casse

Le langage est *case-sensitif* (sensible à la casse)¹ et les accentués sont autorisés en Python 3 (mais pas en Python 2). Ceci signifie que `cout`, `Cout` et `COUT` réfèrent trois mots différents et `coût` est licite.

1. Il fait la distinction entre minuscules/majuscules.

1.2 Les mots-clés



Mots-clés (*keywords*)

Dits aussi **mots réservés**, ils constituent le vocabulaire du langage. Ils forment l'ossature d'un script en définissant la structure de ses données ou de ses instructions.



Propriété

Les mots-clés ont une **signification inchangeable et prédéfinie**, c.-à-d. qu'ils ne peuvent pas être redéfinis.



Liste des mots-clés

(1) Mots-clés (2) Instances

- (1) `and as assert break class continue def del elif else except finally for from global
if import in is lambda nonlocal not or pass raise return try while with yield`
- (2) `None False True`



Environnements IDE

Les mots-clés y sont en gras (ou en couleurs).

1.3 Les identifiants



Identifiant

Dit aussi **identificateur**, c'est un **nom** choisi par le concepteur pour désigner les entités : variable, fonction, type, etc. Cette définition reste valable dans son domaine de validité, à condition qu'elle ne soit pas recouverte par une autre définition. Il **ne doit pas** correspondre à un mot-clé.



Identifiant

Séquence de lettres (A...Z, a...z), de chiffres (0...9), de lettres accentuées ou du caractère souligné (_, *underscore* en terme anglo-saxon). Il doit commencer par une lettre ou un souligné.



Le souligné est significatif

Ainsi `a`, `a_` et `_a` sont des identifiants différents.

1.4 Règles d'écriture d'un identifiant

Le but est de trouver un nom qui soit suffisamment **court** tout en restant **explicite** (parlant) et qui **ne prête pas à confusion**. Voici quelques règles et limitations traditionnelles. (Il est impossible de décrire ici toutes les particularités des différents langages. Reportez-vous aux règles spécifiques de votre langage.)

Identifiant

Suite de caractères alphanumériques **d'un seul tenant** (pas de caractères blancs) et **ne commençant jamais par un chiffre**. Ainsi `x1` est correct mais non `1x`.

Écriture d'un nom composé

On distingue principalement deux notations :

Notation « tiret bas » : Elle utilise le souligné entre les mots (par ex. `note_math`) et non pas le signe moins « - » qui est réservé à la soustraction dans la plupart des langages. Ainsi `note-math` serait interprété comme la soustraction des éléments `note` et `math`.

Notation « chameau » (*camelCase*) : Elle consiste à mettre une majuscule au début des mots, généralement à partir du deuxième (par ex. `noteMath` ou `valeurMaxOuMin`).

Indices et Exposants

Ils sont **proscrits** (par ex. `x1`, `z6`. On les notera `x1`, `z6`).

Mots-clés

Ceux des langages de programmation sont **interdits** (par ex. `for`, `if`, `return`, `while`... pour C/C++, Java, Python) et on déconseille d'utiliser les mots-clés du pseudo-code (tels que `entier`, `autre`, `pour`...).

Caractères exclus

La plupart des langages n'autorisent pas les caractères accentués (tels que à, ç, ê, ï, etc.), ni les caractères spéciaux (tels que &, ', #, etc.), ni les lettres des alphabets non latins (tel que Δ). Enfin certains font la distinction entre les minuscules et majuscules, d'autres non. En algorithmique, nous admettons les caractères accentués du français (par ex. `durée`, `intérêts`, etc.).

Cas du souligné

Les identifiants **commençant** par le soulignement (`_`) sont à éviter car ils sont soit réservés pour les différentes implémentations du langage (en C/C++) ou parce qu'ils ont une signification pour le langage (en Python).

Cas d'une unique lettre

Évitez la lettre `I` ou `i` et la lettre `l` qui se confondent avec le chiffre 1, ainsi que la lettre `0` ou `o` qui se confond avec le chiffre 0.

1.5 Séparateurs et espaces blancs



Espace blanc

Suite de un ou plusieurs caractères choisis parmi : espace, tabulation horizontale, tabulation verticale, changement de ligne ou changement de page.



Séparateurs

Comprend les opérateurs (+, -, *, /...) et les caractères de « ponctuation ».



Quelques séparateurs

	Symbole	Rôle
:	deux-points	début de bloc
()	parenthèses	encadrent des expressions
[]	crochets	symboles du composant liste
.	point	accès à un composant structurée
,	virgule	sépare les éléments d'une liste
;	point-virgule	sépare les instructions
' '	quotes	encadrent les chaînes de caractères
" "	guillemets	encadrent les chaînes de caractères

1.6 Primitives



Bibliothèque

Ensemble de fonctionnalités ajoutées à un langage de programmation. Chaque bibliothèque décrit un thème.



Pour les utiliser

```
import nomBiblio # importe le module nomBiblio
from nomBiblio import desTrucsUtiles # importe uniquement ceux spécifiés
```



Primitives

Noms de fonctions (`abs`, `log`, `sin...`), d'opérateurs (`div`, `mod...`) ou de traitement (`afficher`, `saisir...`). Elles acceptent un ou plusieurs paramètres et jouent le même rôle syntaxique qu'un identifiant.



Appel d'une primitive

```
P(x,...) # procédure
r = F(x,...) # fonction
```

Explication

Appelle (on dit aussi **invoque**) la procédure `P` ou la fonction `F` avec les arguments `x...`. Dans le cas de fonction, la valeur retournée peut être utilisée en tant que macro-expression.

2 Variables, types et valeurs

2.1 Variable, type et valeur



Variable

Élément informatique qu'un script peut manipuler.

Décrite par :

- Un **identifiant** unique qui la désigne.
- Un **type** qui définit de quel « genre » est l'information associée.
- Une **valeur** qui doit respecter le type.



Type d'une donnée

Appelé aussi **domaine de définition**, il spécifie la taille en mémoire occupée par la donnée, quels traitements (opérations) elle peut (et ne peut pas) subir ainsi que l'intervalle de valeurs (minimale et maximale) qu'elle peut prendre.



Utilité du type des données

Il est :

- Utile au concepteur d'algorithmes pour lui permettre de structurer ses idées.
- Très utile au programmeur car il permet de détecter des erreurs potentielles.
- Indispensable au compilateur car les différents types ne sont pas tous représentés de la même façon. La représentation d'un même type peut même varier d'un processeur à l'autre.

2.2 Types intégrés



Types intégrés (*builtins*)

Dits aussi **types de base**, **types fondamentaux** ou encore **types primitifs**, ils correspondent aux données qui peuvent être traitées directement par le langage.



Types intégrés

Domaine	Algorithmique	Équivalent Python
$\mathbb{Z}$	Entier	<code>int</code>
$\mathbb{R}$	Réel	<code>float</code>
$\mathbb{B}$	Booléen	<code>bool</code>
$\mathbb{A}$	Caractère	—
$\mathbb{T}$	Chaîne	<code>str</code>



Python : Le type Caractère

Il n'existe pas : un caractère est simplement une chaîne de longueur 1.

2.3 Littéraux



Littéral

Valeur explicite au sein d'un script.

Il possède *un type* déterminé par sa valeur (entier, réel, caractère...).



Notation scientifique d'un réel

Expression « $m \text{ e } p$ » qui signifie $m \cdot 10^p$ (où m est une suite de chiffres décimaux et p un entier). Ainsi, $12.4e-5 \equiv 12.4 \cdot 10^{-5} \equiv 0.000124$. Elle permet d'exprimer de très petits nombres (ex : $2.5e-201$) et de très grands nombres (ex : $5e156$).



Point décimal

Un réel se distingue d'un entier par l'ajout d'un point (.) à la fin.

Pour la lisibilité, préférez .0 (ex. 5.0 (réel) mais 5 (entier)).



Littéraux

- **Entier** : Suite de chiffres éventuellement préfixé par un signe (+ ou −).
- **Réel** : S'écrit en notation décimale ou en notation scientifique.
- **Booléen** : Ils identifient le **Vrai** (mot-clé **True**) et le **Faux** (mot-clé **False**).
- **Chaîne** : Se place entre quotes (') ou entre guillemets (").

2.4 Exemples : Variables, types et valeurs

Exemples de types

Pour représenter ou décrire :

- Longueur, largeur et surface d'un rectangle : on prendra un réel.
- Nom d'une personne : une chaîne.
- Age d'une personne : un entier.
- Si un étudiant est redoublant ou pas : un booléen.
- Un mois : on préférera un entier donnant le numéro du mois (par ex : 3 pour le mois de mars) plutôt qu'une chaîne (par ex : "mars") car les manipulations et les calculs seront plus simples.

Exemples de littéraux

(Avec leur équivalent mathématique)

Type	Équivalent Mathématique	Littéraux
Entier	$\mathbb{Z}$ (entiers)	3, -5, -100, 122, 0
Réel	$\mathbb{R}$ (réels)	3.1425, -1.12e-10, 0.0
Booléen	$\mathbb{B}$ (booléens)	False, True
Caractère	$\mathbb{A}$ (alphanumériques)	'a', '1', 'A', '&', '#'
Chaîne	T (textes)	"c'est du texte", "a", ""

3 Déclarations

3.1 Déclaration de variables



Déclaration de variables

Consiste à associer un type de données à une ou un groupe de variables. Toute variable doit impérativement avoir été déclarée avant de pouvoir figurer dans une instruction exécutable.



Déclaration de variables

```
nomVar = expression  
nomVar1, nomVar2, ... = expr1, expr2, ...
```

Explication

Déclare des variables d'identifiants `nomVarI` (le nom) de type `TypeVar`. Le type `TypeVar` est défini implicitement (*typage dynamique*) par la valeur de l'`expression`. La fonction intégrée `type(nomVar)` permet de connaître son type.

3.2 Définition de constante



Constante

Littéral à lequel est associé un **identifiant** (par convention, écrit en MAJUSCULES) afin de rendre plus lisible et simplifier la maintenance d'un script. C'est donc une information pour laquelle nom, type et valeur sont figés.



Définition de constante

```
nomConst = expression
```

Explication

Définit la constante d'identifiant `nomConst` de type `TypeConst` et lui affecte une valeur (littéral ou expression) spécifiée. Le type `TypeConst` est défini implicitement par la valeur de l'expression.



Python

Les constantes n'existent pas en PYTHON : nous utiliserons donc des **variables** globales.

3.3 Exemples : Déclarations

Exemple : Déclarations de variables

```
j, k, n = 0, 0, 0
racine1, racine2 = 0.0, 0.0
flag = True
c = 'o'
texte = "..."
```

Ici `j,k,n` sont déclarées comme variables entières, `racine1,racine2` comme variables réelles, `flag` comme variable booléenne, `c` comme variable caractère et `texte` comme variable chaîne de caractères. On aurait aussi bien pu écrire ces déclarations de la façon suivante :

```
j = 0
k = 0
n = 0
racine1 = 0.0
...
```

Cette forme peut s'avérer pratique, notamment si l'on veut commenter les déclarations utilisées, variable par variable.

Exemple : Initialisations de variables

```
PI = 3.14159
MESSAGE = "Appuyez sur une touche pour continuer"
```

`PI` désigne une constante réelle (approximation de π) et `MESSAGE` une constante chaîne de caractères du texte entre les guillemets.

4 Structure générale

4.1 Commentaire



Commentaire (narratif)

Texte qui n'est **ni lu, ni exécuté** par la machine. Il est essentiel pour rendre plus lisible et surtout plus compréhensible un script par un être humain.



Commentaire orienté ligne

```
... # rend le reste de la ligne non-exécutable
```



Commentaire orienté bloc

```
"""  
rend le code entouré non exécutable...  
"""
```

4.2 Instruction composée



Instruction

Ordre donné à l'ordinateur qui a pour effet de changer l'état de la mémoire ou le déroulement du programme ou bien de communiquer avec les unités périphériques (clavier, écran, imprimante, etc.).



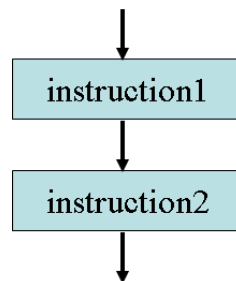
Instruction composée ou Bloc

Regroupement syntaxique de 0, 1 ou plusieurs instructions (et déclarations) comme une unique instruction.



Séquentialité

Les algorithmes et programmes présentés sont exclusivement séquentiels : l'*instruction2* ne sera traitée qu'une fois l'exécution de l'*instruction1* achevée.



Bloc

```
Bloc: #<- deux points
  instruction1 #indentation
  instruction2 #même indentation
  ...
```



Conventions usuelles

Savoir présenter un script, c'est montrer que l'on a compris son exécution.

- Chaque ligne comporte une seule instruction.
Le point-virgule « ; » est le séparateur d'instructions.
- Un bloc est défini par des lignes de **même indentation** (espaces ou tabulations).



Python : L'indentation

C'est une erreur de syntaxe d'indenter deux instructions d'une valeur différente dans un même corps de bloc. De même, ne mélangez pas les tabulations et les espaces car cela donne lieu à des erreurs de compilation qui peuvent paraître incompréhensibles.

**Python : Instruction**

Une instruction se termine à la fin d'une ligne mais elle peut se poursuivre sur plusieurs lignes si ces dernières se terminent par un antislash (`\`), si une paire de `()`, `[]`, `{}` ou une chaîne encadrée de guillemets triples n'est pas refermée. Plusieurs instructions simples peuvent apparaître sur une même ligne si elles sont séparées par un point-virgule (`;`).

4.3 Structure générale



Structure générale

```
from biblio import des_trucs_utiles
déclaration_des_objets_globaux
déclarations_et_définitions_de_fonctions_utiles

def PGPrincipal():
    corps_du_programme

PGPrincipal()
```

Explication

Un script est constitué par :

- Un **en-tête** qui demande d’inclure les modules indiqués.
- Un **corps** lequel contient une fonction (ici `PGPrincipal()`) définie comme étant la fonction « principale » .

Le script commence son exécution sur le bloc de la fonction `PGPrincipal` qui débute après les deux-points, se déroule séquentiellement et se termine sur la fin de l’indentation.



Conseil

On veillera à ce qu’un script tienne sur une vingtaine de lignes (donc, en pratique, sur un écran de 40 x 80 caractères ou une page). Ceci implique que, si votre script devait être plus long, il faudra le découper, comme nous le verrons plus loin.

4.4 Exemple : Le plus petit algorithme



Le plus petit script

```
def PGVide1():  
    pass  
  
PGVide1()
```

Explication

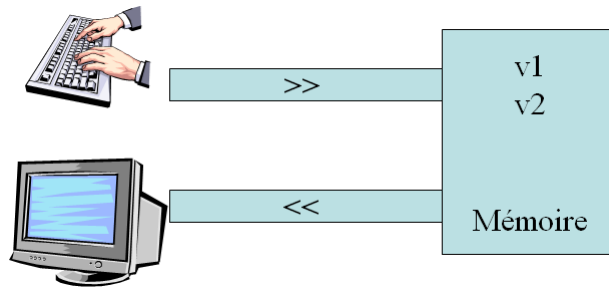
C'est le plus petit script qui ne fait rien.

5 Interactions avec l'extérieur

5.1 Interactions avec l'extérieur

Entrées/sorties clavier/écran

Ce sont les **interactions avec l'extérieur** les plus simples et « naturelles ». Afficher de l'information ou des résultats à destination de l'utilisateur est capital pour la convivialité des programmes. Quant à la saisie, elle permet à l'utilisateur d'entrer les données qui seront stockées dans des variables en vue d'un traitement.



Terminologie

La saisie consiste à affecter une valeur *externe* à une variable : on parle aussi d'**affectation externe**.

5.2 Saisie de données



Saisie de données

```
nomVarI = input(["Invite"]) # lecture d'une chaîne
nomVarI = int(input(["Invite"])) # entrée typée d'un entier
nomVarI = float(input(["Invite"])) # entrée typée d'un réel
nomVarI = bool(input(["Invite"])) # entrée typée d'un booléen
nomVar1, ..., nomVarN = eval(input(["Invite"])) # typage dynamique
```

Explication

Ordonne à la machine de lire des valeurs `valI` depuis le clavier et de les stocker dans les variables `nomVarI` (qui doivent exister c.-à-d. déclarées).



Remarque

Par défaut, ce qui est tapé au clavier est envoyé à l'écran et temporairement placé dans un tampon pour permettre la correction d'erreurs de frappe. On peut donc se servir des touches [←] et [Suppr] pour effacer un caractère erroné ainsi que les flèches [<] et [>] pour se déplacer dans le texte.

5.3 Affichage de résultats



Affichage de résultats

```
print(expr1,...,exprN,end=" ") # SANS retour de ligne  
print(expr1,expr2,...,exprN)  # AVEC retour de ligne
```

Explication

Ordonne à la machine d’afficher les **valeurs** des expressions `exprI`. Par défaut, elles sont séparées par un espace et se terminera par un saut à la ligne. Le paramètre `sep` permet de préciser le séparateur.



Python : print

Avec PYTHON 3x, `print` devient une fonction : il faut donc mettre des parenthèses autour de ce que l’on souhaite afficher contrairement à PYTHON 2x.



Variables devant être initialisées

On ne peut *afficher* que des expressions dont les variables qui la composent ont été affectées préalablement.

5.4 Exemples : Communication des données/résultats



Exemple : Afficher un mot

```
def PGAfficher1():  
    print("Bonjour")  
  
PGAfficher1()
```

Explication

Ce programme affiche le mot « Bonjour ».



Exemple : Saisir un nombre

```
def PGSaisir1():  
    x = float(input("Tapez un nombre reel au clavier? "))  
    print("==> Vous avez tape ", x, ", felicitations", sep="")  
  
PGSaisir1()
```

Explication

Ce programme permet à l'utilisateur de saisir un réel par `float(input())` et de le mémoriser dans la variable `x` puis de le ré-afficher par `print`.

6 Expressions arithmétiques

6.1 Qu'est-ce qu'une expression ?



Expression, opérandes, opérateurs

Éventuellement accompagnés de parenthèses, une **expression** est une séquence « bien formée » (au sens de la syntaxe) d'**opérandes** (littéraux, variables ou expressions) et d'**opérateurs** destinée à l'évaluation.

Exemples

- Dans l'expression $1+2$, les valeurs 1 et 2 sont des opérandes et $+$ est un opérateur.
- Dans $(a+b)*c$, l'opérande de gauche de l'opérateur $*$ est $(a+b)$.
- $(2*(13-j)/(1+4))$ est une expression numérique BF (bien formée).
- $(3+(5*(-2))$ est non BF (parenthésage incorrect).
- $3 (5) *)-2($ est non BF (opérateur et parenthésage incorrect).



Pluralité d'un opérateur

Un opérateur :

- **unaire** (ou monadique) porte sur un opérande (ex : le $-$ dans l'expression $-a$).
- **binaire** (ou dyadique) sur deux opérandes (le plus fréquent).
- **ternaire** (ou triadique) sur trois opérandes.

Un opérateur à n opérandes est dit **n -aire**.

6.2 Opérateurs arithmétiques



Opérateurs arithmétiques

Dits aussi **opérateurs algébriques**, ils agissent sur des opérandes de type numérique.



Opérateurs arithmétiques

Opérateur Mathématique	Signification	Équivalent Python
+	(unaire) valeur	<code>+a</code>
-	(unaire) opposé	<code>-a</code>
+	addition	<code>a + b</code>
-	soustraction	<code>a - b</code>
*	multiplication	<code>a * b</code>
/	division décimale	<code>a / b</code>
div	division entière	<code>a // b</code>
mod	modulo (reste de la division entière)	<code>a % b</code>
^	élévation à la puissance	<code>a ** b</code>



Python : Division euclidienne

Sous PYTHON 2, la division `/` appliquée à des entiers retourne le résultat de la division entière. Sous PYTHON 3, l'opérateur `/` donne toujours le résultat de la division réelle et l'opérateur `//` donne le résultat de la division entière.



Division euclidienne, cas des négatifs

Il n'y a pas unicité du quotient et du reste lorsque le dividende ou le diviseur sont négatifs. Si a et b sont deux entiers relatifs dont l'un au moins est négatif, il y a plusieurs couples (q, r) tels que $a = b \times q + r$ avec $|r| < |b|$. Par exemple, si $a = -17$ et $b = 5$ alors $(q = -3, r = -2)$ ou $(q = -4, r = 3)$ sont deux solutions possibles. Habituellement, q et r sont choisis comme le quotient et le reste de la division entière de $|a|$ et $|b|$ affectés du signe approprié (celui permettant de vérifier $a = b \times q + r$ avec $|r| < |b|$). Dans l'exemple ci-avant, la solution retenue serait $(q = -3, r = -2)$. Par contre, la norme impose que la valeur de $(a \text{ div } b) * b + a \text{ mod } b$ soit égale à la valeur de a .



Division par zéro

Tout emploi de la division devra être accompagné d'une réflexion sur la valeur du dénominateur, une division par 0 entraînant toujours l'arrêt d'un script.

6.3 Évaluation des expressions



Évaluation d'une expression

Elle conduit à la **valeur littérale** correspondant au **résultat** de l'expression lequel s'appuie sur un *système de priorités* relatives entre les opérateurs et des *règles d'associativité*. La **priorité** (ou **préséance**) fait référence à l'ordre d'application des opérateurs et l'**associativité** peut être exprimée de droite à gauche ou de gauche à droite. Les parenthèses permettent de modifier ou de souligner la priorité.



Ordre de priorité des opérateurs arithmétiques

Comme en mathématique :

1. Les opérateurs unaires (+, -) (priorité la plus élevée)
2. L'opérateur d'exponentiation (^) (s'il existe)
3. Les opérateurs multiplicatifs (*, /, div, mod)
4. Les opérateurs additifs (+, -) (priorité la plus basse)

La règle d'associativité s'applique en cas d'ambiguïté entre opérateurs du même ordre de priorité.



Opérateur mod

Préférez de parenthéser vos expressions (ne serait-ce que pour la lisibilité!).

Pour des entiers :

- $a * b \text{ mod } c \equiv (a * b) \text{ mod } c$
- $a \text{ mod } b * c \equiv (a \text{ mod } b) * c$
- $a + b * c \text{ mod } d \equiv a + ((b * c) \text{ mod } d)$

Exemple : Écrivez $(a * b) \text{ mod } c$ plutôt que $a * b \text{ mod } c$.



Exponentiations

Deux exponentiations successives sont évaluées en commençant par celle de droite.

Ainsi $a^b{}^c$ signifie a^{b^c} mais $a/b/c$ représente $\frac{a}{b \cdot c}$.

6.4 Exemples : Opérateurs arithmétiques

Exemple : Cas de deux entiers

Soient *a* et *b* deux variables entières valant respectivement 10 et 3. Le tableau illustre les opérations arithmétiques qui peuvent leur être appliqués :

Opération	Expression	Valeur
addition	<i>a</i> + <i>b</i>	13
soustraction	<i>a</i> - <i>b</i>	7
multiplication	<i>a</i> * <i>b</i>	30
division entière	<i>a</i> // <i>b</i>	3
modulo	<i>a</i> % <i>b</i>	1

Exemple : Cas de deux réels

Soient *a* et *b* deux variables réelles valant respectivement 12.5 et 2.0. Le tableau illustre les opérations arithmétiques qui peuvent leur être appliqués :

Opération	Expression	Valeur
addition	<i>a</i> + <i>b</i>	14.5
soustraction	<i>a</i> - <i>b</i>	10.5
multiplication	<i>a</i> * <i>b</i>	25.0
division	<i>a</i> / <i>b</i>	6.25

Exemple : Priorité entre opérateurs arithmétiques

(1)

(2)

(1)

(1)

(2)

(0)

a * b

-

c / d

mod e

+

f

(a * b)

-

(c / d mod e)

+

(f)

((a * b) - (c / d mod e))

+

(f)

((a * b) - ((c / d) mod e))

+

(f)

2

3

1

4

5

6

D'abord la

« gestion

des

priorités »

suivi des

« évaluations »

6.5 Cohérence de type numérique



Règle de promotion

Pour qu'une opération numérique binaire (+, -, *, /) puisse s'effectuer, il faut que ses deux opérandes soient du **même type** ou d'un **type compatible**. Lorsque ce n'est pas le cas, préalablement à l'opération, il y a **promotion** de l'opérande de type le plus faible vers le plus grand.

Exemple

Si le réel `r` vaut 1.0 et l'entier `j` vaut 2, l'expression `r/j` se calcule ainsi :

1. Promotion de l'entier 2 en réel 2.0
2. Évaluation du calcul de 1.0/2.0 qui donne le réel 0.5

6.6 Fonctions mathématiques



Fonctions mathématiques

Elles agissent sur des paramètres à valeurs réelles et donnent un résultat réel.



Pour les utiliser

```
import math
```



Quelques Fonctions mathématiques

Fonctions Mathématiques	Signification	Équivalent Python
π	valeur du nombre π	<code>math.pi</code>
$\text{acos}(x)$	Angle (exprimé en radians) dont le cosinus est égal à x	<code>math.acos(x)</code>
$\text{asin}(x)$	Angle (exprimé en radians) dont le sinus est égal à x	<code>math.asin(x)</code>
$\text{atan}(x)$	Angle (exprimé en radians) compris entre $-\pi/2$ et $\pi/2$ dont la tangente est égale à x	<code>math.atn(x)</code>
$\lceil x \rceil$	Réel de l'entier supérieur	<code>math.ceil(x)</code>
$\cos(x)$	Cosinus de x (exprimé en radians)	<code>math.cos(x)</code>
e^x	Exponentielle de x (base e)	<code>math.exp(x)</code>
$ x $	Valeur Absolue de x	<code>math.fabs(x)</code>
$\lfloor x \rfloor$	Réel de l'entier inférieur à x	<code>math.floor(x)</code>
$\log(x)$	Logarithme naturel (base e)	<code>math.log(x)</code>
x^y	Puissance	<code>math.pow(x,y)</code>
$\sin(x)$	Sinus de x (exprimé en radians)	<code>math.sin(x)</code>
$\sqrt{x}$	Racine carrée du nombre positif x	<code>math.sqrt(x)</code>
$\tan(x)$	Tangente de x (exprimé en radians)	<code>math.tan(x)</code>



Racine carrée

Attention de ne l'utiliser qu'avec un radicant positif.

7 Affectation interne

7.1 Affectation interne



Affectation interne

Opération qui **fixe une valeur** à une variable.



Affectation interne

```
nomVar = expression
```

Explication

Place la valeur de l'**expression** dans la zone mémoire de la variable de nom **nomVar**. En algorithmique, le symbole **<-** (qui se lit « devient ») indique le sens du mouvement : de l'expression située à droite **vers** la variable à gauche.



Rappel

Toutes les variables apparaissant dans l'**expression** doivent avoir été affectés préalablement. Le contraire provoquerait un arrêt de l'algorithme.



Conversions implicites

Il est de règle que le résultat de l'expression à droite du signe d'affectation soit de même type que la variable à sa gauche.

7.2 Exemples : Affectation interne

Exemples : Affectations correctes

```
a = 3
x = y
delta = math.sqrt((b*b - 4*a*c)/(2.0*a))
test = True
```

La première instruction affecte la valeur 3 à la variable `a`, la seconde la valeur de `y` à la variable `x`, la troisième la valeur de l'expression arithmétique à la variable `delta` et la dernière la valeur logique à la variable `test`.

Exemples : Affectations incorrectes

```
somme + 1 = 3;
somme = 3n;
```

La première est invalide car `somme+1` n'est pas une variable et la deuxième car `3n` n'est ni un nom correct de variable ni une expression correcte.

7.3 Exemple : Affectations et conversions

Ce script emploie des fonctions mathématiques. Il montre l'affectation et les conversions implicites de type dans les expressions.



Script @[pgaffect1.py]

```
import math

def PGAffect1():
    x = 6.0
    y = x * -3
    print("En A : y vaut ", y, " et x vaut ", x, sep="")
    x = (x - 4) ** 2
    print("En B : y vaut ", y, " et x vaut ", x, sep="")
    x = math.sin(math.sqrt(x * -y + 9))
    print("En C : y vaut ", y, " et x vaut ", x, sep="")

PGAffect1()
```

Explication

Il déclare des variables réelles `x` et `y` et est constitué de quatre instructions d'affectation.

- La première affecte la valeur 6 à la variable `x`. A l'issue de cette instruction, `x` vaut le réel 6.0 (conversion implicite de l'entier 6 en réel).
- La deuxième est l'affectation d'une expression algébrique à une variable réelle. L'expression est d'abord reconnue comme un produit à effectuer portant sur une valeur réelle et une valeur entière. Comme les types des valeurs sont différents, la valeur -3 est d'abord convertie en 3.0 puis multipliée avec la valeur de la variable `x` (définie) qui vaut 6.0. Ainsi, la multiplication a ses deux opérandes réels, elle est effectuée et la valeur réelle -18.0 est affectée à la variable `y`.
- La troisième évalue d'abord le paramètre `x-4` de la fonction qui vaut $6.0-4=2.0$ puis appelle la fonction `carré` (opérateur puissance) qui donne la valeur 4.0 laquelle modifie la valeur de la variable `x`. L'ancienne valeur de `x`, qui était 6.0, est perdue. Le déroulement séquentiel fait naturellement « oublier » les instructions effectuées en ne conservant que les valeurs courantes des variables.
- La quatrième suit le même principe : elle évalue d'abord `x*-y+9` qui vaut $4.0*18.0+9=81.0$ laquelle est transmise à la fonction `racinecarrée` (`sqrt`) qui donne 9.0, laquelle est transmise à la fonction `sin` qui donne 0.4121184852.

Résultat d'exécution

```
En A : y vaut -18.0 et x vaut 6.0
En B : y vaut -18.0 et x vaut 4.0
En C : y vaut -18.0 et x vaut 0.4121184852417566
```

8 Trace d'exécution d'un script

8.1 Trace d'exécution

Faire la trace d'exécution d'un script consiste à suivre pas-à-pas (instruction par instruction) sur une feuille de papier, le contenu des variables et les valeurs entrées et sorties. C'est un excellent moyen pour :

- Comprendre sa fonctionnalité.
- Tester sur quelques exemples que les opérations sont programmées correctement. Notez qu'une trace s'effectue sur un script de syntaxe correcte.

Préparation de la trace

La trace d'exécution, exercice facile dans son principe, demande beaucoup de rigueur et d'attention. Pour réussir une trace d'exécution :

1. Prenez une feuille de papier et créez un tableau.
2. Réservez une colonne pour les valeurs en entrée. Placez-y les valeurs dans l'ordre dans lequel l'exécution de l'algorithme prévoit de les faire entrer. Cette colonne sera appelée *colonne-des-valeurs-en-entrée*.
3. Réservez une colonne pour les valeurs en sortie. Les valeurs sorties lors de l'exécution par l'instruction d'affichage seront inscrites dans l'ordre dans lequel l'exécution les fait sortir. Cette colonne sera appelée *colonne-des-valeurs-en-sortie*.
4. Réservez une colonne pour chaque variable déclarée. Intitulez-la par son nom. Nous appellerons une telle colonne, une *colonne-de-variable*.
5. Exécutez votre script, instruction par instruction, en commençant par la première instruction.

Trace d'exécution d'une instruction saisir

La trace d'exécution d'une instruction `saisir(XXX)`, où `XXX` est le nom d'une variable, se passe en deux temps :

1. Dans la *colonne-des-valeurs-en-entrée*, barrez d'un trait horizontal la prochaine valeur à entrer. Ce faisant, les valeurs inscrites sur la *ligne-des-valeurs-en-entrée* sont biffées au fur et à mesure des exécutions des instructions `saisir`. Lors de la première exécution d'une instruction `saisir`, biffez la première valeur, lors de la deuxième exécution d'une instruction `saisir`, biffez la deuxième valeur...
2. Inscrivez sur la *colonne-de-variable* de `XXX` la valeur entrée (celle que vous venez de biffer dans la *colonne-des-valeurs-en-entrée*). Si une valeur se trouve déjà inscrite sur la colonne-de-variable de `XXX`, biffez cette valeur d'un trait horizontal. Chaque fois que le contenu d'une variable est modifié par une instruction, procédez de cette façon : biffure de l'ancienne valeur et inscription de la nouvelle valeur. Ainsi pour connaître la valeur mémorisée dans une variable, repérez la seule valeur non biffée de sa *colonne-de-variable*.

Trace d'exécution d'une instruction afficher

La trace d'exécution d'une instruction `afficher(XXX)`, où `XXX` est le nom d'une variable ou une valeur, se réduit à recopier sur la *colonne-des-valeurs-en-sortie*, la valeur mémorisée dans `XXX` ou directement la valeur. C'est-à-dire avec notre système de biffure, la seule valeur non barrée de la *colonne-de-variable* `XXX`. **Attention !** La valeur mémorisée par la variable `XXX` ne change pas, il ne faut surtout pas la biffer.

Trace d'exécution d'une instruction d'affectation

La trace d'exécution d'une instruction `XXX<--expr`, où `expr` dénote une expression, conduit à effectuer dans l'ordre :

1. L'évaluation de l'expression `expr` à droite du symbole d'affectation (`<--`). C'est le calcul en remplaçant chacune des variables par les valeurs qu'elles contiennent. Pour connaître les valeurs contenues dans les variables, reportez-vous aux *colonne-de-variable* de chacune des variables.
2. Remplacez le contenu de la variable `XXX` indiquée à gauche du symbole `<--` par la valeur trouvée au calcul en (1). Plus précisément, biffez la valeur se trouvant éventuellement sur la *colonne-de-variable* de `XXX` et inscrivez-y, la valeur trouvée.

8.2 Exemple complet de programme / pgdiffsecs



Énoncé

Calculer le nombre de secondes écoulés entre deux instants de la journée. En entrée : deux instants sous la forme heures, minutes, secondes. Supposer que le deuxième est le plus grand. En sortie : le nombre de secondes écoulées entre les deux instants. Exemple : 17, 23, 41, 21, 19, 14 comme valeur en entrée donnera après exécution 62621, 76754, 14133 comme valeur en sortie.

Algorithme

L'algorithme que nous proposons est :

- Saisir le premier instant (dans trois entiers `hr`, `mm`, `ss`).
- Calculer le nombre de secondes écoulées depuis minuit de cet instant dans `nsecs` comme suit :
 - Multiplier l'heure `hr` par le nombre de `SECONDES_PAR_HEURE`.
 - Multiplier les minutes `mm` par le nombre de `SECONDES_PAR_MINUTE`.
 - Ajoutez les secondes `ss` à la somme des résultats des deux multiplications ci-dessus.
- Afficher le résultat de l'addition `nsecs`.
- Saisir le deuxième instant (dans `hr`, `mm`, `ss`).
- Calculer le nombre de secondes écoulées depuis minuit de cet instant dans `nsecs2` comme ci-avant.
- Afficher le résultat de l'addition `nsecs2`.
- Faire la différence entre `nsecs2` et `nsecs`.
- Afficher le résultat de la soustraction.



Script @[pgdiffsecs.py]

```
SECONDES_PAR_MINUTE = 60
SECONDES_PAR_HEURE = 3600

def PGDiffsecs():
    print("hr mm ss?", sep="", end="")
    hr = int(input())
    mm = int(input())
    ss = int(input())
    nsecs = hr * SECONDES_PAR_HEURE + mm * SECONDES_PAR_MINUTE + ss
    print(nsecs)
    print("hr mm ss?", sep="", end="")
    hr = int(input())
    mm = int(input())
    ss = int(input())
    nsecs2 = hr * SECONDES_PAR_HEURE + mm * SECONDES_PAR_MINUTE + ss
    print(nsecs2)
    nsecs = nsecs2 - nsecs
    print(nsecs)

PGDiffsecs()
```

Explication

Le programme est une traduction de l’algorithme. Il entre six nombres, les trois premiers nombres représentant le premier instant de la journée sous la forme heures, minutes et secondes. Les trois suivants représentent un autre instant, également sous la forme heures, minutes et secondes. Il sort trois nombres représentant respectivement le nombre de secondes écoulées entre minuit et le premier instant, le nombre de secondes écoulées entre minuit et le second instant et la différence en secondes entre ces deux instants.

Trace d’exécution

Voici la trace d’exécution avec pour valeur en entrée : 17, 23, 41, 21, 17, 14.

SECONDES_PAR_MINUTE : 60						
SECONDES_PAR_HEURE : 3600						
Valeurs en entrée : <u>17</u> ^{M1} <u>23</u> ^{M2} <u>41</u> ^{M3} <u>21</u> ^{M6} <u>17</u> ^{M7} <u>14</u> ^{M8}						
	hr	mm	ss	nsecs	nsecs2	Valeurs en sortie
Début	?	?	?	?	?	
M1	<u>17</u>					
M2		<u>23</u>				
M3			<u>41</u>			
M4				<u>62621</u>		
M5						62621
M6	24					
M7		17				
M8			14			
M9					76634	
M10						76634
M11				14013		
M12						14013

8.3 Erreur à l'exécution

Il peut arriver que l'exécution d'un script ne puisse se faire jusqu'au bout. Dans un tel cas, il faut impérativement arrêter sa trace d'exécution et déclarer qu'il y a erreur à l'exécution : il est incorrect ! il faut le corriger.

Au moins cinq cas peuvent se présenter :

Cas 1 : plus de valeurs à entrer Cette erreur se rencontre lorsqu'un script demande d'entrer une valeur dans une variable, et que toutes les valeurs de la ligne *valeurs-en-entrée* sont biffées : il n'y a plus de valeurs à entrer.

Cas 2 : toutes les valeurs n'ont pas été entrées Cette erreur se rencontre lorsqu'un script se termine et qu'il reste une ou plusieurs valeurs de la ligne *valeurs-en-entrée* qui n'ont pas été biffées : toutes les valeurs n'ont pas été entrées.

Cas 3 : incohérence des types Cette erreur se rencontre lorsqu'un script demande d'entrer une valeur dans une variable, et que le type de la prochaine valeur à entrer ne correspond pas au type de la variable désignée par l'instruction *saisir*. Exemple : l'instruction est *saisir(V)* avec *V* une variable numérique et la prochaine valeur à entrer est une chaîne de caractères. Il y a incohérence des types.

Ou encore il demande de faire une affectation et le type du résultat de l'évaluation de l'expression à droite du symbole *<--* n'est pas le même que celui de la variable à gauche du symbole *<--* qui doit mémoriser le résultat. Il y a aussi incohérence des types.

Notez que lorsqu'un script est compilé avant d'être exécuté, ce type d'erreur est détecté dès la compilation avant même l'exécution.

Cas 4 : utilisation d'une variable qui ne contient pas de valeur Cette erreur se rencontre lorsqu'une instruction veut utiliser une variable (soit par l'instruction *afficher*, soit par l'instruction d'affectation) alors qu'aucune valeur ne lui a été encore affectée. Rappel : en début d'exécution aucune valeur n'est affectée aux variables. Pour qu'une variable contiennent une valeur (définie), il faut qu'une instruction *saisir* ou d'affectation y mémorise une valeur.

Cas 5 : résultat erroné A la fin de son exécution, toutes les valeurs en entrée doivent être barrées et sur la ligne réservée aux valeurs en sortie, vous devez trouver les valeurs prévues en sortie. Sinon, considérez que le résultat obtenu n'est pas le bon.



Remarque

L'intérêt de faire une ou deux traces d'exécution après avoir écrit un script n'est pas de montrer qu'il est correct, mais simplement de relever les erreurs d'étourderie et de logique.

8.4 Méthode

Remarquez qu'une trace d'exécution est une activité dynamique ! Elle se construit au fur et à mesure de l'exécution de votre script. Cela implique de l'exécuter pas-à-pas et de noter, au fur et à mesure, les modifications dans les différentes lignes de la trace.

Voici la méthode élémentaire d'élaboration d'un script :

- Faites attention au problème posé. Notamment ayez à l'esprit les tenants (c.-à-d. les valeurs en entrée) et les aboutissants (les valeurs en sortie) de votre script.
- Résolvez les exercices proposés.
- Transcrivez votre analyse en un script.
- Soumettez votre script écrit à une ou deux traces d'exécution.

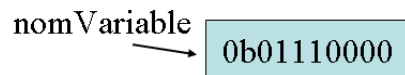
Si la trace obtenue n'est pas celle attendue, réfléchissez à nouveau et modifiez votre script en conséquence pour que la trace d'exécution obtenue soit conforme à celle attendu.

9 Compléments

9.1 Variables et constantes

Emplacement mémoire, unicité des noms

Une variable désigne un **emplacement mémoire** qui permet de stocker une valeur. L'adresse de cet emplacement mémoire est associée au nom de la variable. Manipuler le nom de la variable c'est faire référence à son adresse en mémoire : c'est le système qui réalise cette association.



Puisqu'un script exploite généralement plusieurs variables pour traiter les informations, chacune est désignée par un nom unique. L'**unicité des noms** de variables permet à la machine d'identifier précisément la variable à être manipulée.

Variable

Une variable peut être vue comme une boîte dans laquelle est rangée une information qui peut être récupérée à tout moment. À la différence de la boîte qui redevient vide lorsqu'on en retire son contenu, la variable stocke une « copie » des données qu'un algorithme y range (via l'affectation). Ainsi, lorsque l'algorithme récupère la valeur d'une variable, il récupère une copie du contenu de la variable.

Une variable conserve ses données (c.-à-d. son contenu) jusqu'à ce qu'une autre valeur y soit stockée, remplaçant la valeur antérieure, ou jusqu'à ce que la variable soit détruite par l'algorithme.

Initialisation par défaut

Elle consiste à donner une valeur initiale de façon automatique aux variables déclarées (par exemple 0 pour les variables numériques, **Faux** pour les booléens, etc.). Celle-ci n'existe ni en algorithmique, ni dans les langages JAVA ou PYTHON. En C/C++, cette initialisation diffère selon que cette variable est locale ou globale (cf. @[Règles de visibilité] dans [Algorithmes paramétrés (2)]).

Erreurs à éviter

Voici une synthèse :

- Une variable ou une constante déclarée avec un certain type ne peut pas être déclarée une deuxième fois au même niveau.
- Il faut déclarer la variable ou la constante avant de pouvoir l'utiliser.
- Une constante doit être initialisée lors de sa déclaration.
- Il faut attribuer une valeur à la variable avant d'utiliser sa valeur.

Variables en programmation

Quelques conseils :

- Déclarez toujours une variable, même si certains langages de programmation peuvent les déclarer pour vous. Le type pris par défaut ne sera pas forcément le plus adapté.
- Déclarez les variables en début de programme si vous souhaitez un lexique clairement localisé. Sinon déclarez-les au plus près de leur première utilisation.
- Initialisez toujours les variables (par une affectation interne ou externe). Certains langages de programmation et/ou compilateurs les initialisent automatiquement à zéro, d'autres pas. En particulier, le compilateur JAVA émet le message d'erreur suivant dans un tel cas :

```
Variable may not have been initialized
```

Prenez le principe général qu'elle n'est **jamais** initialisée par défaut.

9.2 Affectation : Compléments



Affectation multiple

```
nomVar1, nomVar2, ... = expr1, expr2, ...
```

Explication

Effectue les affectations chaînées de la droite vers la gauche.



Notations abrégées

```
x @= y # <=> x = x @ y avec @ parmi {+, -, *, /, //, %}
```

Explication

Notations abrégées des opérateurs arithmétiques pour l'affectation.

9.3 Transtypage



Transtypage (*casting*)

Conversion explicite du type d'une expression vers un autre type compatible.



Transtypage

```
T(expr)
```

Explication

Convertit le type de l'expression `expr` vers le nouveau type `T`.

Avec `T` parmi : `str`, `int`, `float`, `bool` (Nous verrons les autres types dans les modules suivants).