Table des matières

Algorithmique et programmation 1

Préambule

Bienvenue dans le cours **Algorithmique et Programmation 1** !

Pour qui ? Pour quoi ?

Ce cours est destiné en particulier aux étudiant.e.s ayant peu d'expérience en programmation.

Il va couvrir les bases de la programmation *impérative*, à travers le langage de programmation *Python* (3), ainsi que constituer une introduction à la discipline (et à la pensée) algorithmique.

Ces notes sont pensées pour accompagner, enrichir et détailler les transparents présentés en *cours magistral* (CM). Les transparents sont aussi disponibles en consultation en ligne ou au téléchargement.

Elles sont en grande partie basées sur les notebooks des années précédentes, rédigées principalement par Antoine Meyer.

Bonne découverte et bon semestre !

Marie et Léo

Trouver des informations

- **Contacts** (à chercher dans l'annuaire) :
  - Responsable de formation : Antoine Meyer
  - Secrétaire de formation : Ramatoulaye Barry ramatoulaye.barry@univ-eiffel.fr (absence, pb admin, …)
  - Responsables de l'UE : Marie van den Bogaard et Léo Exibard
- **Page web** :
  - https ://elearning.univ-eiffel.fr/course/view.php ?id=9175
  - Ressources diverses (ouvrages, liens, annonces)
  - Sujets des TD, des TP, liens vers les supports de cours
- **Communication** :
  - Discord : https ://discord.gg/6jrfmUwcSp
  - Mail : *exclusivement* sur votre adresse <vous@edu.univ-eiffel.fr>
  - *Webmail* : http ://partage.univ-eiffel.fr

Programme et références

Ce cours recouvre très largement la partie *Langages et programmation* du programme de l'option de spécialité NSI de première générale proposée par les (des) lycées français. Certaines notions des parties *Algorithmique*, *Représentation des données* et *Histoire de l'informatique* seront aussi abordées. (cf. https ://www.education.gouv.fr/media/23690/download)

Beaucoup de ressources sont disponibles pour apprendre et approfondir les concepts présentés dans ce cours. Nous recommandons notamment le livre *Informatique :Inf*, collection *Fluorescience*s, éditions *Dunod*, partie 1 *Mathématiques pour l'Informatique* et partie 2 *Algorithmique et Programmation*, disponible à la bibliothèque universitaire Georges Pérec. Le livre pdf gratuit de G. Swinnen, *Apprendre à programmer avec Python 3*, dont le lien se trouve sur la page e-learning du cours, peut aussi s'avérer pertinent pour le lecteur curieux.

En (très) résumé, voici les notions abordées dans la suite de ces notes et pendant les séances de CMs, TDs et TPs :

- Notion d'**algorithme** : définition, intérêt, exemples, mise en oeuvre
- Briques de base de la **programmation impérative** : variables, assignation, expressions, opérations, structure de contrôle, représentation des données (types simples, types construits, structures de données), fonctions
- Outils mathématiques et de raisonnement : bases de numération, arithmétique "euclidienne" (division entière, reste, modulo, ppcm, pgcd), booléens et valeurs de vérités
- Machinerie Python : syntaxe, fonctionnement de la mémoire, méthodes utiles, listes, dictionnaires

— Bonnes pratiques de programmation : Règles de style, Prototypage, Découpage en sous-tâches, Documentation, Tests
— (en bonus, si le temps le permet : une introduction au concept de *récursivité*)

Introduction

Une histoire de zéros et de uns…

ou *du binaire aux langages de programmation*

ou *comment en est-on arrivé là ?*

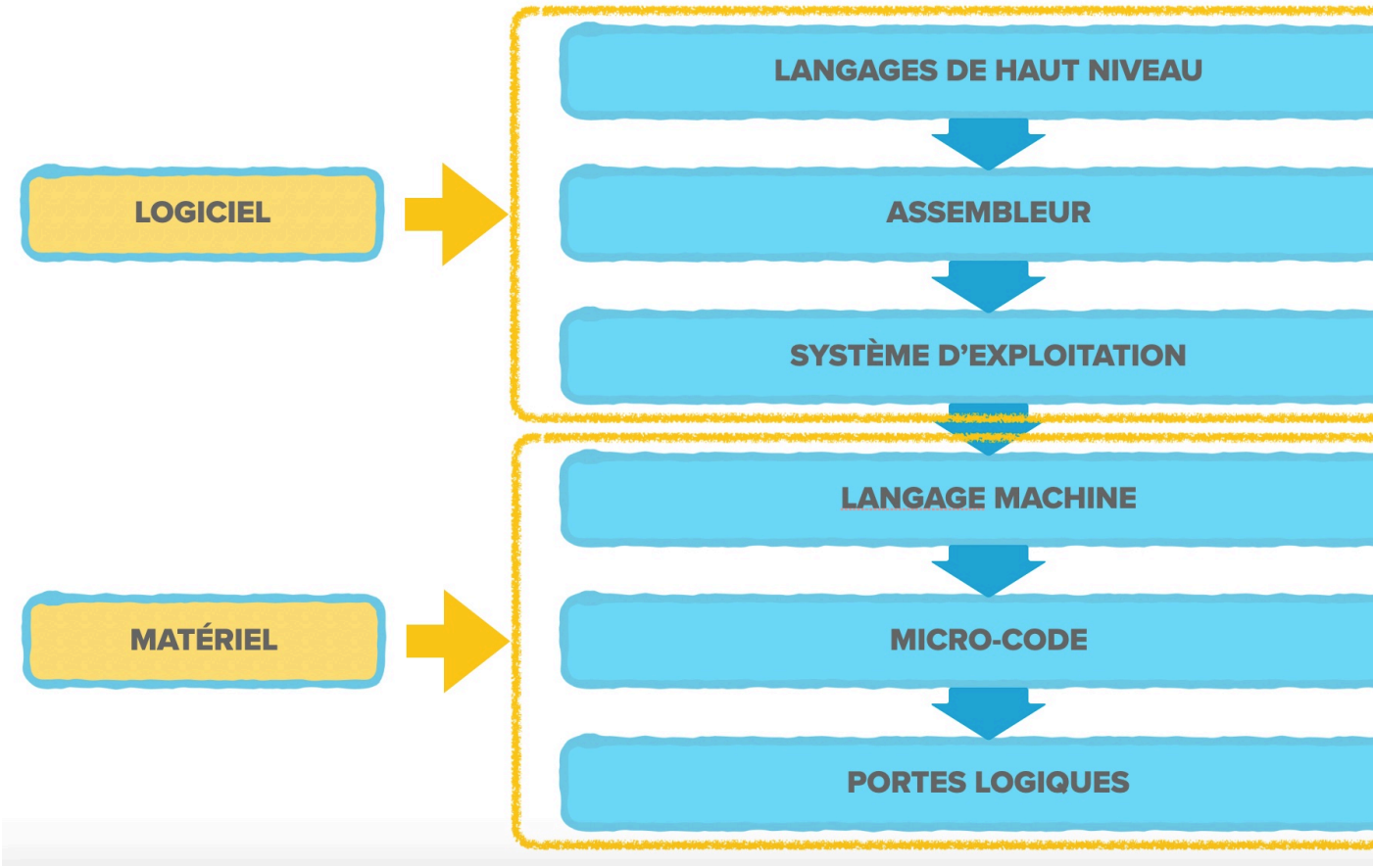
Dans cette section, nous allons essayer d'éclairer l'écart entre le monde des *bits* et le monde des langages de programmation *haut niveau*, comme le Python. La majorité des gens ont en effet une vague idée du fait que l'*informatique*, ou même l'*ordinateur*, "ça marche avec des *zéro*s et des *uns*".

Ce petit morceau de vulgarisation scientifique dans la culture générale, bien que raisonnable, est parcellaire et très éloigné de la réalité de nos interactions avec l'informatique. Réalité quotidienne d'une grande partie de l'humanité : utilisation d'applications sur nos ordinateurs, téléphones et montres pour réaliser tout un tas de tâches (communication, divertissement, administratif, apprentissage, graphisme…), mais aussi réalité quotidienne pour une partie beaucoup plus restreinte de l'humanité : les *informaticien.ne.s* !

Concrètement, depuis très longtemps (à l'échelle de l'histoire de l'informatique), les informaticien.ne.s ne manipulent pas de 0 et de 1 (de *bits*) "à la main" (ou alors très rarement, on verra cela). En particulier, en programmation, on utilise en pratique des langages qu'on appelle de *haut niveau* : des langages qui sont relativement compréhensibles à la lecture par un être humain initié. On comprend donc mieux les programmes, et on les écrit aussi plus facilement.

D'accord, mais on nous avait donc menti avec ces histoires de zéros et de uns ?

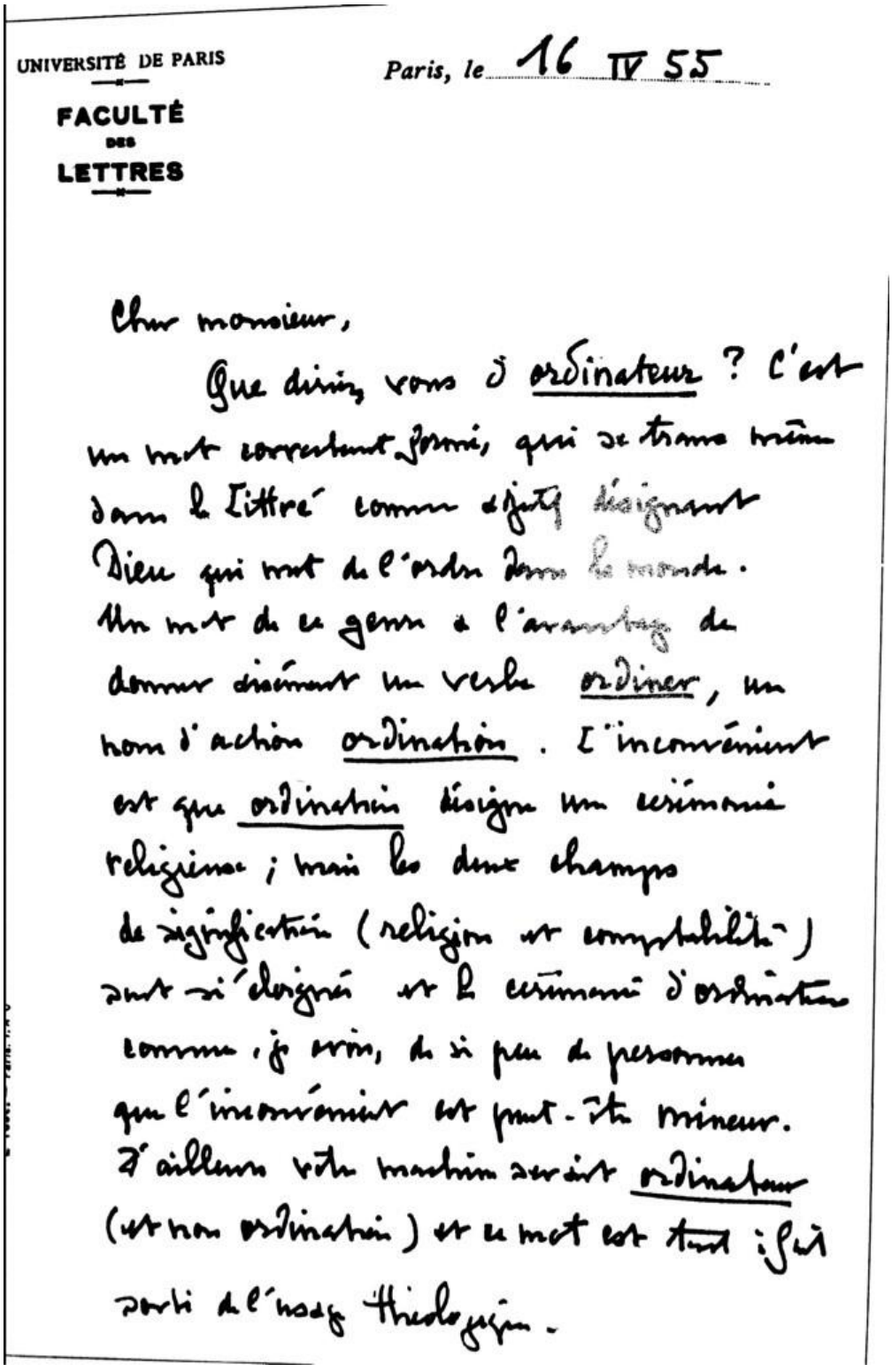
Non ! Ils sont toujours là, mais entre eux et ce que nous programmons, il y a plusieurs couches, ou strates, de traductions (et un peu d'électronique). Nous ne rentrerons pas dans les détails dans ce cours (les UE d'architecture et de compilation le feront en partie), mais voici une figure qui représente ces différentes strates.



Nous allons donc nous intéresser à la couche supérieure de ce schéma, celle des langages de programmation de haut niveau. Vous savez déjà probablement qu'il en existe plusieurs, et en fait, une *multitude* (dont le Python). Disons-en un peu plus sur cette multitude avant de passer au contenu algorithmique et technique.

De l'universel au particulier

On a vu superficiellement qu'à partir de la strate "langage machine", tout s'exprimait effectivement en binaire. C'est vrai pour les données, mais aussi pour les programmes ! Or, il est impossible pour le cerveau humain de lire/comprendre/écrire une telle diversité de signifiants encodés avec un alphabet aussi petit (on le rappelle, seulement deux lettres, `0`et `1`). Surtout quand on sait qu'un des objectifs de l'informatique, c'est de traiter beaucoup d'information, ou effectuer beaucoup de calculs, de manière automatique et rapide. Très vite, les langages de programmation ont vu le jour : en 1954, le bal commence avec **Fortran**. C'est un an *avant* le choix du mot **ordinateur** pour parler de *computer* en français.



UNIVERSITÉ DE PARIS

FACULTÉ DES LETTRES

Paris, le 16 IV 55

Cher monsieur,

Que dirin vous à ordinateur ? C'est un mot correctement formé, qui se trouve même dans le Littré comme adjectif désignant Dieu qui met de l'ordre dans le monde. Un mot de ce genre a l'avantage de donner directement un verbe ordiner, un nom d'action ordination. L'inconvénient est que ordination désigne une cérémonie religieuse ; mais les deux champs de signification (religion et comptabilité) sont si éloignés et la cérémonie d'ordination comme, je crois, de si peu de personnes que l'inconvénient est peut-être mineur. D'ailleurs votre machine devrait ordinateur (et non ordinatrice) et ce mot est tout à fait sorti de l'usage théologique.

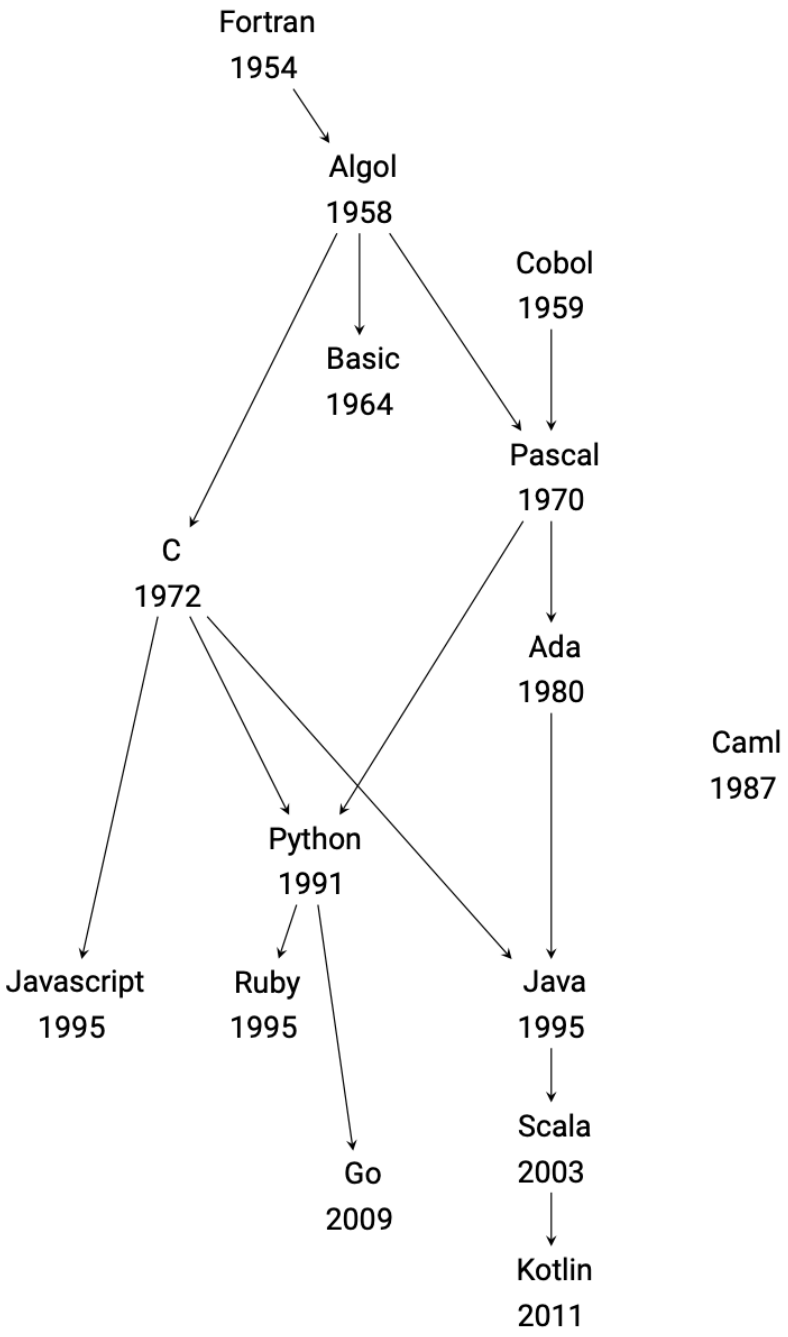
Puis la création de nouveaux langages s'accelère, chacun ayant ses propres spécificités qui le rendent plus ou moins adapté à telle ou telle application. Par exemple, le *COBOL*, créé en 1959, est particulièrement robuste et fiable pour manipuler des grandes quantités de données structurées, et il est donc encore utilisé à ce jour dans les banques, les assurances et certaines administrations.

Le langage *C*, quant à lui, permet de contrôler de manière assez fine l'utilisation de la mé-

moire, et ainsi d'être très performant et économe en ressources. Cette liberté d'usage de la mémoire le rend peut-être moins accessible aux débutants, et même un peu dangereux si l'on est pas prudent ! En revanche, il permet justement d'apprendre la rigueur et d'approfondir certains concepts fondamentaux de programmation : c'est pourquoi vous vous plongerez dedans dès la L2, si vous choisissez le cursus informatique à l'issue de la L1.

Enfin, le **Python** : né en 1991, il est beaucoup utilisé pour mettre en place des programmes rapidement. Il est (relativement) simple et peu verbeux, ce qui le rend particulièrement adapté à l'apprentissage de la programmation et à l'écriture de *scripts* (programme simple souvent conçu pour automatiser des tâches élémentaires). C'est lui qui est notre langage de référence dans ce cours, et que l'on va s'employer à maîtriser de plus en plus.

La figure ci-dessous montre un échantillon de la généalogie des langages de programmation :



Un mot sur les *paradigmes* de programmation

— À votre avis, pourquoi est-ce que le *Caml* est tout seul dans son coin ?

La réponse vague : à cause des *paradigmes* de programmation. Cette expression un peu terrifiante veut simplement dire qu'il existe plusieurs *style* de programmation, différentes manières de concevoir les programmes.

Ainsi, dans la programmation **impérative**, on va indiquer une série d'instructions à exécuter étape par étape. Intuitivement, on explique *quoi* faire et *comment* en donnant des instructions précises. Les premiers langages de programmation sont dans ce style, mais pas seulement ! Le **C** et le **Python** sont des exemples de langages qui peuvent mettre en oeuvre ce

paradigme. En fait, la plupart des langages contiennent ce qu'il faut pour programmer de manière impérative, et on peut voir ce paradigme comme l'ensemble des briques de bases de l'algorithmique et de la programmation. C'est d'ailleurs pour cette raison que nous allons explorer cette façon de programmer dans ce cours.

On peut aussi citer la programmation **orientée objet**, dans laquelle on est centré sur le concept d'*objet* qui a des propriétés et des méthodes pour agir et interagir avec l'extérieur et avec lui-même. Le **Java** (que vous verrez en L3) et le **C++** (que vous pourrez voir en master) sont des langages de ce type. On peut aussi programmer en orienté objet avec Python, mais cela ne sera pas abordé dans ce cours.

Enfin, la programmation **fonctionnelle** est elle centrée sur le concept de *fonction* : très proche des mathématiques, cette façon de penser peut être pertinente pour écrire des programmes très élégants. Le Caml fait partie de cette famille de langages, tout comme le Haskell que vous pourrez aborder en L3.

Briques de base d'algorithmique

Voici un aperçu des briques de base, qui vont nous occuper pour tout le semestre :

Un programme informatique traite des **données** pour en extraire de l'information. Elles peuvent être déjà stockées quelque part, ou bien fournies par l'utilisateur auquel cas on parle d'**entrées** (avec le clavier, la souris, etc). Le programme peut également interagir avec l'utilisateur, par exemple en affichant quelque chose à l'écran ou en commandant l'impression d'un document (les **sorties**).

Penchons-nous maintenant sur la manière dont le programme traite les données : essentiellement, un programme consiste en une suite d'**instructions**, qui peuvent être répétées ou encore exécutées en fonction de certaines conditions, conformément aux **structures de contrôle** du programme (instructions conditionnelles, boucles `for` et `while`). Enfin, certaines parties du programme peuvent être isolées dans des **fonctions** pour être réutilisées et améliorer la lisibilité du code.

À quoi ça correspond en Python ?

Tout cela est peut-être un peu abstrait. Nous approfondirons chaque notion au fil de l'année, mais voici quelques indications pour vous donner une idée, à la lumière de ce que vous avez vu et verrez en TD et en TP (et éventuellement de ce que vous savez déjà de Python) : - Données : c'est l'entrée du programme, par exemple les variables globales et les informations stockées dans des fichiers - Entrées-sorties : vous pouvez demander une entrée à l'utilisateur via la fonction `input()`, et afficher des éléments à l'écran avec `print()`. Nous verrons des méthodes plus élaborées (par exemple via des interfaces graphiques) avec `fltk`. - Opérations : les opérations arithmétiques `+`, `-`, `*`, `/`, etc ; logiques `and`, `or`, `not` ; la concaténation `+` ; et tout un tas d'opérations plus compliquées que nous verrons au fil de l'année. - Fonctions : vous avez déjà pu manipuler `input` et `print` pour les entrées-sorties, ainsi que `int` et `str` pour convertir en entier et en chaîne de caractère, nous en verrons de nombreuses autres ! - Structures de contrôle : il s'agit des instructions conditionnelles `if ... else ...`, des boucles `for ... in ...` et des boucles `while ...:`.

Mais pas de panique, nous allons voir tout cela en détail en CM !

Un peu plus sur Python

C'est parti pour expliciter les premières briques dans le langage Python. La suite est un catalogue présentant les outils et notions pratiques à connaître pour comprendre et écrire de premiers programmes en Python.

Types de valeurs

Toutes les valeurs en Python possèdent un **type**. Le type d'une valeur définit les **opérations** possibles. Les types de base sont :

- les nombres entiers (`int`) ou décimaux (`float`)
- les booléens (`bool`)
- les chaines de caractères (`str`)
- un type de valeur indéfinie (`NoneType`)

Nombres entiers (`int`)

```
6
```

```
6
```

```
12345
```

```
12345
```

```
-4  # un entier négatif
```

```
-4
```

```
2 ** 1000  # un très très grand entier
```

```
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376
```

```
0b101010  # un entier en binaire
```

```
42
```

```
0x2a  # un entier en hexadécimal
```

```
42
```

Nombres décimaux (`float`)

```
3.14
```

```
3.14
```

```
-1.5
```

```
-1.5
```

```
3 * .1  # un nombre décimal qui ne "tombe pas juste"
```

```
0.30000000000000004
```

```
12.
```

```
12.0
```

```
4.56e3  # notation scientifique
```

```
4560.0
```

Booléens (`bool`) Ce type permet de représenter les deux valeurs de vérité « vrai » et « faux ».

```
True  # vrai
```

```
True
```

```
False  # faux
```

```
False
```

Attention : Les majuscules/minuscules sont importantes :

```
true  # provoque une exception (une erreur)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

Cell In[14], line 1
----> 1 true  # provoque une exception (une erreur)


NameError: name 'true' is not defined
```

Chaînes de caractères (`str`) Une chaine de caractères est une succession de symboles (lettres, chiffres, ou autres) entre guillemet

```
'bonjour'  # guillemets simples
```

```
"hello !"  # guillemets doubles
```

```
"⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠   ."  # caractères non-latins
```

```
# Chaînes longues
"""Ce plat est supposé être dégusté au petit-déjeuner
mais convient aussi comme dessert. Les pancakes sont
traditionnellement accommodés avec du sirop d'érable
et une noix de beurre mais rien n'empêche de les
dévorer au sucre, au jus de citron ou avec de la pâte
à tartiner."""
```

Il faut faire un peu attention pour écrire une chaîne de caractères contenant des apostrophes ou des guillemets :

```
"King's Landing"
```

```
'Mon nom est "Personne".'
```

Caractères spéciaux : `\n, \t, \', \", \\`

```
"sauts\nde\nligne"
```

```
print("sauts\nde\nligne")
```

```
# tabulation, touche 
print("Du\tsur\ntexte\t2 colonnes")
```

```
print("D'autres symboles spéciaux : \' \" \\")
```

Valeur indéfinie (NoneType)

```
None  # ça a l'air inutile mais en fait c'est bien pratique
```

Opérations

Le type d'un objet détermine les **opérations** qu'on peut lui appliquer.

C'est un principe *très important* en Python.

Opérations sur les nombres Addition (`a + b`), soustraction (`a - b`), multiplication (`a * b`), puissance (`a ** b`) - sur deux **`int`** et produisant un **`int`** - ou sur deux **`float`** et produisant un **`float`** - ou sur un **`int`** et un **`float`** et produisant un **`float`**

```
4 + 5
```

```
4 - 5.5
```

```
4. * 5.
```

```
4 ** 2
```

```
4 ** 0.5
```

Division "réelle" (`a / b`) : produit toujours un **`float`**

```
1 / 3  # valeur approchée !
```

```
4 / 2  # ne donne pas un entier !
```

Division euclidienne : - **quotient** : `a //b` - **reste**, ou **modulo** : `a % b` - les deux en même temps : `divmod(a, b)` - si `a` et `b` de type **`int`**, produisent un **`int`**, sinon un **`float`**

```
7 // 2
```

```
7 % 2
```

```
divmod(7, 2)
```

```
4 // 2  # cette fois c'est un entier...
```

```
4 % 2  # le reste est nul car 4 est pair (divisible par 2)
```

```
4.0 // 1.75  # donne un float !
```

```
4.0 % 1.75
```

Les opérations suivent les règles de priorité usuelles :

```
4 + 2 * 1.5
```

On peut aussi utiliser des parenthèses :

```
(4 + 2) * 1.5
```

Opérations sur les chaînes de caractères Concaténation : `s + t`

```
'Gustave' + 'Eiffel'
```

```
'Gustave' + ' ' + 'Eiffel'
```

Répétition : `s * a`

```
'Hip ' * 3 + 'Hourra !'
```

```
('Hip ' * 3 + 'Hourra ! ') * 2
```

Beaucoup d'autres opérations (sur les chaînes, les nombres…) : *on verra ça plus tard*

 Le sens de * et + n'est pas le même sur les chaînes et sur les nombres !

Conversions / transformations de type

On a parfois besoin de convertir une valeur d'un type à l'autre

— N'importe quel objet en chaîne avec la fonction `str`

```
"J'ai " + 10 + ' ans.'
```

```
"J'ai " + str(10) + ' ans.'
```

— Un **`float`**, ou *parfois* un `str` en **`int`**

```
int(3.5)  # float vers int
```

```
int('14')  # str vers int
```

```
int('3.5')  # impossible : deux conversions (str -> float -> int
    )
```

```
int('deux')  # impossible : ne représente pas un nombre
```

— Un **`int`**, ou *parfois* un `str` en **`float`**

```
float(3)  # int vers float
```

```
float('14.2')  # str vers float
```

```
float('3,5')  # impossible : virgule au lieu de point
```

```
float('bonjour')  # impossible : ne représente pas un nombre
```

Déterminer le type d'une expression

Grâce à la fonction prédéfinie `type`

```
type("salut")
```

```
type(4 / 2)
```

```
type(2 * 4.8)
```

Exercice : valeur et type d'une opération Pour chacune des instructions suivantes : 1. donner le type et le résultat de l'expression donnée ; 2. vérifier le résultat.

On pourra utiliser la fonction `type` si nécessaire pour vérifier le type du résultat.

```
2 * 5
```

```
2 + 1.5
```

```
2.0 * 4
```

```
'2.0' * 4
```

```
'2.0' * 4.0
```

```
4 / 2
```

```
4.0 / 2
```

```
5 / 2
```

```
5 % 2
```

```
5 // 2
```

```
int(4.0) / 2
```

```
str(4) / 2
```

```
'toto' + str(4)
```

```
float(4) * 2
```

```
int(str(4) * 2)
```

```
'toto' + 'titi'
```

```
int('toto') + 'titi'
```

```
int(2.0) * 4
```

```
'toto' * str(4)
```

```
int('1.25')
```

Variables et affectations

Une **variable** est un *nom* servant à désigner une valeur - Une variable est remplacée par sa valeur dans les calculs - Seules les opérations du type de la valeur sont permises

L'**affectation** est le fait de lier une *valeur* à une *variable* - Syntaxe : `nom` = *une expression*
Attention : Ce n'est pas *du tout* le = des mathématiques, il faut le lire comme "prend la valeur"

```
x = 3
y ='UGE'
z = x + 2
x, y, z
```

— On peut *réaffecter* une variable (même avec une valeur d'un type différent)

```
x
```

```
x = 'UGE'
```

```
x
```

— On ne peut utiliser une variable que si elle a été préalablement définie !

```
foo
```

Modèle de mémoire de Python

Modèle de mémoire : une image simplifiée de la manière dont fonctionne la mémoire de l'interpréteur Python

Deux zones principales : - la zone des données (le « tas », en anglais *heap*) - la zone des espaces de noms (la « pile », en anglais *stack*)

Dans Python Tutor : pile à gauche, tas à droite

Le tas Le tas est comme un *très* gros cahier dans lequel sont décrits les objets manipulés par un programme - Chaque objet décrit dans le cahier commence à un certain numéro de page, qu'on appelle son *adresse* - Certaines pages sont blanches, d'autres sont remplies

La pile La pile est comme l'index du cahier

— À chaque variable est associé le numéro de page d'un objet
— Un groupe de variables et les numéros de page correspondants est appelé **espace de noms**
— La pile contient l'espace de noms **global**, contenant les noms définis par nos programmes
(en réalité la pile contient aussi d'autres espaces de noms, on en reparlera)

```
%%nbtutor -r -f
x = 3
y ='UGE'
z = x + 2
```

La notion d'état L'**état** de l'interpréteur pendant l'exécution c'est

— le numéro de la ligne suivante à exécuter dans le programme
— le contenu de diverses variables internes de l'interpréteur
— le contenu de la pile (donc tous les espaces de noms)
— le contenu du tas (donc toutes les données du programme)

à un moment donné. Les **instructions** modifient généralement l'état.

Étapes d'une affectation

Affectation simple

```
1 x = 40 + 2
```

1. Évaluation de l'expression à droite du = (ici 42)

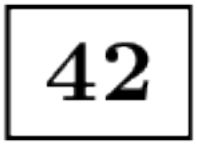
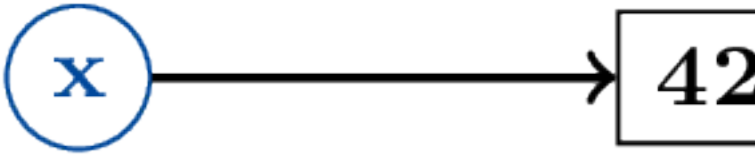


FIGURE 1 – affectation1.png

La valeur 42 de type `int` est stockée dans le tas *(écrite sur une page du cahier)*

2. Création du nom x dans l'espace de noms (sauf s'il existe déjà)
   On ajoute x à la pile *(on ajoute une ligne pour x à l'index du cahier)*



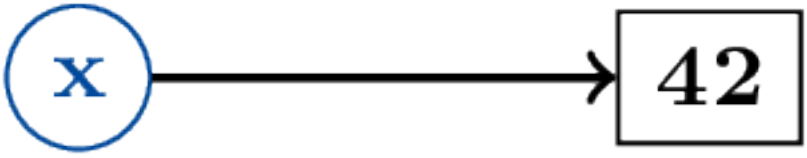
3. Création du lien entre *variable* et *valeur*
   L'adresse de l'objet 42 est associée à la variable x *(on écrit dans l'index le numéro de la page contenant l'objet 42 à x)*

Deuxième exemple

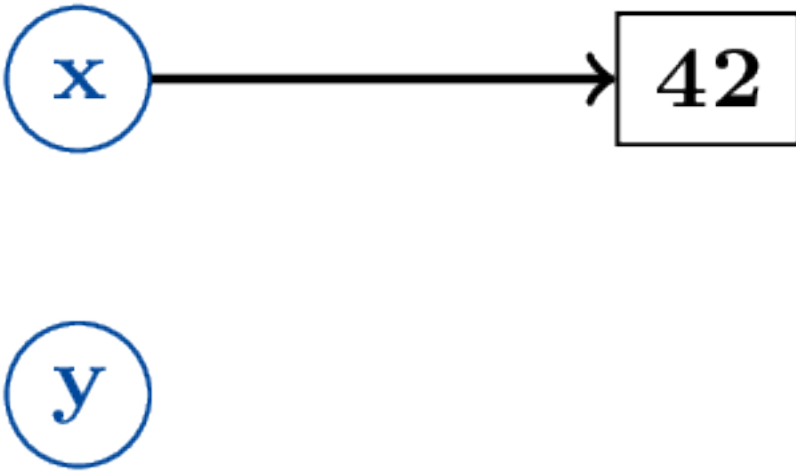
```
1 y = x
```

Dans cet exemple le x en partie droite de l'affectation désigne la **valeur** actuellement associée à la variable x !

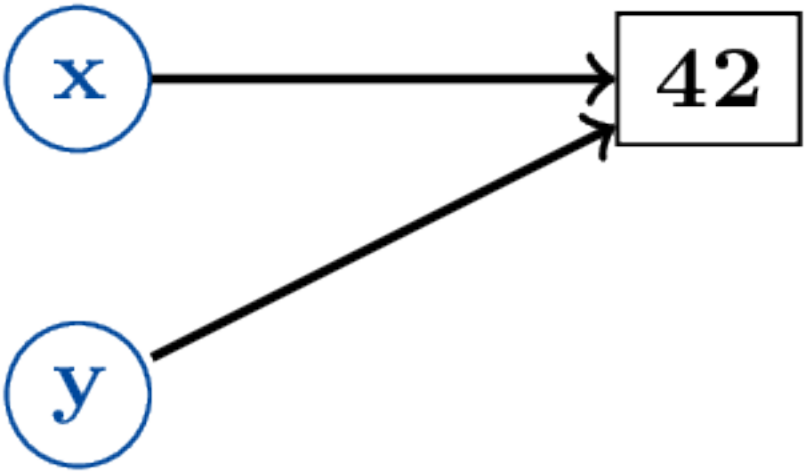
1. Calcul du *membre droit* après remplacement de chaque variable par la valeur associée



(ici x remplacé par 42)



2. Création du nom y (sauf si déjà créé)



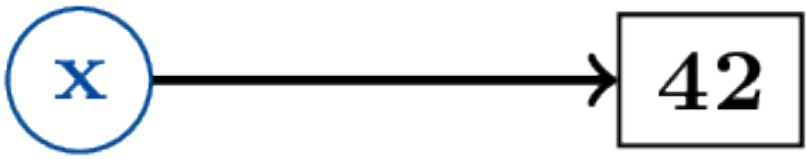
3. Création du lien entre y et 42

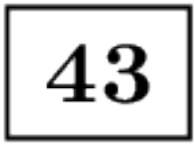
Troisième exemple

```
x = x + 1
```

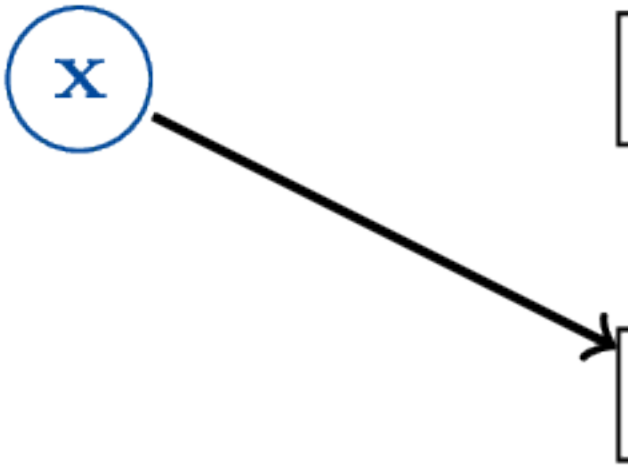
Dans cet exemple, le x de gauche désigne le nom x lui-même, mais celui de droite désigne la **valeur** actuellement associée à x

1. Calcul du *membre droit* après remplacement de chaque variable par la valeur associée





(ici x remplacé par 42, résultat : 43)



2. Nom x déjà existant, création du lien entre x et 43

Piège ! Que vaut maintenant y ?

```
y
```

Même si x et y désignaient avant le même objet (la même adresse, le même *numéro de page*), changer la page que désigne x n'a aucun effet sur y ! On n'a pas *modifié* l'objet 42, qui est toujours là !

```
%%nbtutor -r -f
x = 40 + 2
y = x
x = x + 1
```

Remarque : L'instruction `x = x + 1` peut aussi s'écrire `x += 1`. On appelle cela une **incrémentation** de x.

```
x += 1
```

De même, `x *= 2` est une version plus concise de `x = x * 2`.

Dernier exemple

```
x = -6.5
```

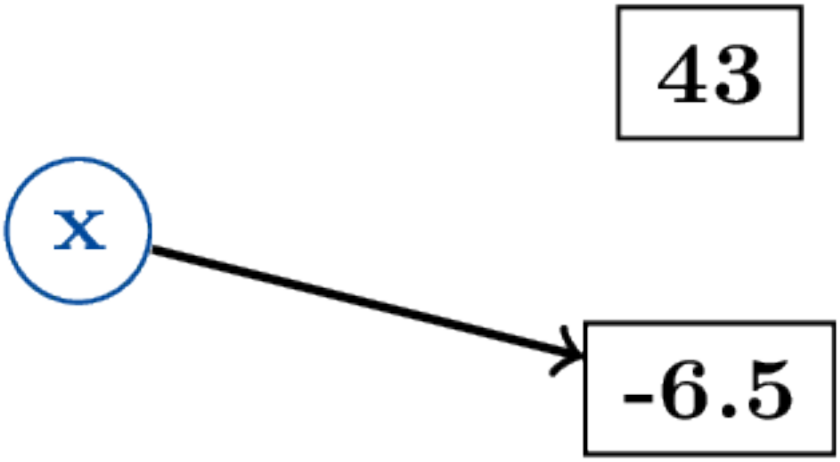


FIGURE 2 – affectation8.png

Quelques points de détail

- On peut réaffecter une valeur de type différent à une variable (comme dans le dernier exemple)
- En cas de réaffectation, le lien précédent est oublié
- Quand aucun lien n'existe vers un objet, il est "détruit" par le ramasse-miettes ou *garbage collector* (la page est effacée !)

Exercice : état de la mémoire après une suite d'affectations Dessiner l'état de la mémoire à l'issue des instructions suivantes :

```
x = 2
y = 3
x += y
y *= 2
```

Nommage des variables Règles de nommage des variables :

- Commencent par une *lettre*, suivie de *lettres et de chiffres*
- Le caractère *underscore* `'_'` est considéré comme une lettre
- Éviter les caractères spéciaux (accents, cédille, etc.)
- Les *mots réservés* (ou mots-clés) de Python sont interdits

— Il y a aussi des **conventions** (*vues plus tard*)

Exemples : `_ex2` `Ex2mpl1`

Contre-exemple : `2024eiffel`

Mots-clés et autres mots réservés Les mots suivants sont **réservés** pour le langage :

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

https ://docs.python.org/fr/3/reference/lexical_analysis.html#keywords

Exercice : nommage de variables Indiquer parmi les mots suivants ceux qui ne sont pas des noms valides pour une variable :

```
bonjour                 Hi!                 au revoir
                  oui
Ciao                    NON                 byeBye7
                    6hello6
abc                     def                 6hello6
                    _upem_
good_morning            __repr__            good-afternoon
                f()
```

Saisie et affichage

Fonction de saisie : `x = input("Veuillez rentrer ...")`

— L'utilisateur tape une ligne au clavier
— La ligne est stockée sous forme de chaîne de caractères (`str`)
— Cette valeur peut ensuite être affectée à une variable (ici `x`)
— Le message d'invite pour l'utilisateur est facultatif

https ://docs.python.org/fr/3/library/functions.html#input

```python
nb_personnes_ref = 2
nb_convives = input("Combien de personnes ? ")
rapport = nb_convives / nb_personnes_ref
```

```python
nb_personnes_ref = 2
# on convertit immédiatement le texte saisi en int :
nb_convives = int(input("Combien de personnes ? "))
```

```
4 rapport = nb_convives / nb_personnes_ref
```

Fonction d'affichage : `print(x)`

- Affiche dans le terminal la chaîne de caractères associée à `x`
- On peut afficher plusieurs valeurs à la suite : `print(x, y, z, ...)`
- Appelle automatiquement la fonction `str` sur chacun de ses arguments
- S'il y a plusieurs arguments, insère automatiquement des espaces
- Passe automatiquement à la ligne

https ://docs.python.org/fr/3/library/functions.html#print

```
1 nb_personnes_ref = 2
2 # on convertit immédiatement le texte saisi en int :
3 nb_convives = int(input("Combien de personnes ? "))
4 rapport = nb_convives / nb_personnes_ref
5 print("Je multiplie toutes les quantités par", rapport)
```

Remarque : *Il existe de nombreuses possibilités pour l'affichage de texte, consulter la documentation officielle pour plus de détails.*

Structures de contrôle

Les programmes et algorithmes les plus simples consistent à exécuter des instructions les unes après les autres, en **séquence**. C'est néanmoins très vite limité : il arrive fréquemment qu'on ait envie d'agir d'une certaine façon dans un cas et d'une autre dans un autre. Typiquement, on voudrait pouvoir continuer le programme de manière adaptée à une entrée de l'utilisateur.

On va donc s'intéresser aux structures dites *conditionnelles*. Ces structures permettent de "*brancher*" dans le code en fonction de l'évaluation d'une condition, que l'on exprime sous forme d'**expression booléenne**.

Faisons donc d'abord un point sur ce type très important d'expressions :

Expressions booléennes

Les instructions conditionnelles sont écrites à l'aide d'**expressions booléennes**, c'est à dire d'expressions qui s'évaluent en un valeur de type `bool` (`True` ou `False`).

Elles peuvent contenir des opérateurs de comparaison, des opérateurs logiques, etc.

Opérateurs de comparaison

```
a < b   # a strictement inférieur b
a <= b  # a inférieur ou égal à b
a >= b  # a supérieur ou égal à b
a > b   # a strictement supérieur à b
```

— a et b sont des **expressions**
— elles doivent s'évaluer en des valeurs **de même type** (sauf exceptions)

Les opérateurs de comparaison fonctionnent sur de nombreux types de valeurs - Sur les `int` et `float` : ordre habituel sur les nombres - Sur les `str` : ordre *lexicographique* (dictionnaire) - Sur d'autres types qu'on verra plus tard

 **Rappel :** on ne peut pas ordonner des valeurs de types différents (sauf des nombres) !

Égalité ou inégalité

```
a == b  # a égal à b
a != b  # a différent de b
```

— a et b sont des **expressions**
— elles peuvent être de types différents

Les opérateurs == et ! = acceptent des opérandes de types différents - renvoie généralement `False` si les opérandes sont de types différents - sauf parfois entre nombres

 **Attention !** Ne pas confondre l'opérateur d'égalité (==) avec la syntaxe de l'affectation (=) !

```
17 % 2 == 1
```

```
a = -5
a != abs(a)
```

```
1.0 == 3 - 2  # l'égalité fonctionne aussi avec les float...
```

```
0.3 == 3 * 0.1  # mais réserve parfois des surprises
```

Les opérateurs == et ! = acceptent des opérandes de types différents - renvoie généralement `False` si les opérandes sont de types différents - sauf parfois entre nombres

```
2 == '2'
```

```
'bonjour' != None
```

```python
2 == 2.0  # Cas particulier : vrai car float(2) == 2.0
```

 **Attention !** Ne pas confondre l'opérateur d'égalité (==) avec la syntaxe de l'affectation (=) !

Opérateurs logiques

On peut combiner plusieurs expressions booléennes `a` et `b` à l'aide d'opérateurs logiques, inspirés de la logique mathématique.

On peut résumer le comportement de ces opérateurs à l'aide de tableaux, appelés **tables de vérité**.

Négation L'expression `not a` vaut `True` si `a` s'évalue en `False`, et `False` sinon (correspond à $\neg a$).

a	not a
True	False
False	True

Conjonction L'expression `a and b` vaut `True` si `a` et `b` s'évaluent toutes les deux en `True`, et `False` sinon (correspond à $a \wedge b$).

a	b	a and b
True	True	True
True	False	False
False	True	False
False	False	False

Disjonction L'expression `a or b` vaut `True` si `a` s'évalue en `True` ou `b` s'évalue en `True`, et `False` sinon (correspond à $a \vee b$).

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

```
not (3 + 4 != 7)
```

```
4 < 1 or 'Bonjour' >= 'Au revoir'
```

En réalité les opérateurs `and` et `or` ont un comportement un peu spécial appelé **évaluation séquentielle** : on n'évalue le deuxième opérande que si c'est nécessaire pour déterminer le résultat.

- `a and b` est à peu près équivalent à `b if a else a`
- `a or b` est à peu près équivalent à `a if a else b`

Cela signifie qu'on n'évalue pas toujours `b` dans `a and b` et dans `a or b`. Par exemple :

```
1/0  # erreur : division par 0
```

```
True or 1/0 == 1    # ne provoque pas d'erreur !
```

```
False and 1/0 == 1  # ne provoque pas d'erreur !
```

En Python, presque tout objet possède une valeur de vérité et peut s'utiliser comme un booléen… mais les règles sont un peu complexes.

Par exemple :

- les nombres égaux à 0 sont interprétés comme "faux"
- la chaîne vide est interprétée comme "faux"
- la valeur `None` est interprétée comme "faux", etc.

Tout le reste est interprété comme "vrai"

Si on combine ces deux aspects du langage, ça peut donner des choses assez surprenantes…

```
'patate' and 'courgette'
```

```
'courgette'
```

```
'patate' or 'courgette'
```

```
'patate'
```

Ce comportement est largement **hors programme** et non exigible !

Quelques règles utiles

Lois de De Morgan

- dire “non (a ou b)” revient à dire “(non a) et (non b)”
- dire “non (a et b)” revient à dire “(non a) ou (non b)”

En Python :

```
not (a and b) == (not a) or (not b)  # vrai pour tous a et b
not (a or b) == (not a) and (not b)  # vrai pour tous a et b
```

Distributivité

- la conjonction est distributive sur la disjonction
- la disjonction est distributive sur la conjonction

En Python :

```
a and (b or c) == (a and b) or (a and c)  # vrai pour tous a, b et c
a or (b and c) == (a or b) and (a or c)   # vrai pour tous a, b et c
```

Commutativité

- `a and b` est (presque) équivalente à `b and a` (mais elle change l’ordre d’évaluation)
- `a or b` est (presque) équivalente à `b or a` (idem)

Absorption

- `a or True` est (presque) équivalente à `True`
- `True or a` est équivalente à `True`
- `a and False` est (presque) équivalente à `False`
- `False and a` est équivalente à `False`

Invariance

- `a and True` est (presque) équivalente à `a`
- `True and a` est équivalente à `a`

— `a or False` est (presque) équivalente à `a`
— `False or a` est équivalente à `a`

Égalité et négation

— `not a == b` est équivalent à `a != b`
— `not a != b` est équivalent à `a == b`

Comparaisons et opérateurs logiques

— `a < b` est équivalent à `a <= b and a != b`
— `a <= b` est équivalent à `a < b or a == b`
— `a > b` est équivalent à `a >= b and a != b`
— `a >= b` est équivalent à `a > b or a == b`

Comparaisons et négation

— `not a < b` est équivalent à `a >= b` ou encore `b <= a`
— `not a <= b` est équivalent à `a > b` ou encore `b < a`
— `not a > b` est équivalent à `a <= b` ou encore `b >= a`
— `not a >= b` est équivalent à `a < b` ou encore `b > a`

```
x = -1
inf, sup = 0, 10
x >= inf and x <= sup
```

```
x = -1
inf, sup = 0, 10
x < inf or x > sup
```

```
x = 1
inf, sup = 0, 10
not (x < inf or x > sup)
```

```
x = 12
inf, sup = 0, 10
not (x >= inf and x <= sup)
```

Si £ et ¥ représentent des opérateurs de comparaison,

```
exp1 £ exp2 ¥ exp3
```

est une abréviation de

```
exp1 £ exp2 and exp2 ¥ exp3
```

Enchaînement de comparaisons

En Python on peut rassembler plusieurs comparaisons successives en une seule expression (c'est impossible dans de nombreux autres langages).

```
x, inf, sup = 1, 0, 10
inf <= x <= sup
```

```
a, b, c = 1, 1, 2
a == b == c
```

Attention, cela mène parfois à des écritures un peu bizarres…

```
x, inf, sup = 1, 4, 10
inf < sup > x  # ???
```

```
a = 1
b = 2
c = 1
a != b != c
```

Exercice : multiple de 3 ou 5

Étant donnée une variable `x` désignant un nombre, 1. écrire deux expressions booléennes différentes qui valent `True` si `x` est un multiple de 3 et de 5 et `False` sinon, 2. écrire deux expressions booléennes différentes qui valent `True` si `x` n'est un multiple ni de 3 ni de 5 et `False` sinon.

```
x = 10
x % 3 == 0 and x % 5 == 0
```

```
x = 7
not(x % 3 == 0) and not(x % 5 == 0)
```

Exercice

Donner le résultat des expressions logiques suivantes pour les valeurs indiquées de `a`, `b` et `c`

```
a, b, c = 10, 2, 6
a < b or a > c
```

```
a, b, c = 10, 2, 6
a + b < 2 * c
```

```
a, b, c = 10, 2, 6
a - b == b + c
```

```
a, b, c = 10, 2, 6
(a > b and a > c) or (b > a and b > c)
```

```
a, b, c = 10, 2, 6
a < b < c
```

```
a, b, c = 10, 2, 6
a == b == c
```

```
a, b, c = 10, 2, 6
(a <= b and a <= c) or not (b < a)
```

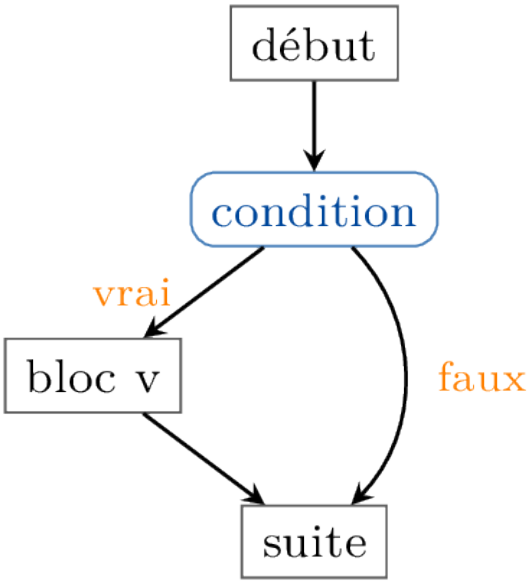
```
a, b, c = 10, 2, 6
not (a > b and a > c) or (b > a and b > c)
```

Instructions conditionnelles

Cas simple : Conditionnelle `Si`

On peut maintenant modifier le flot d'instructions selon la valeur d'expressions booléennes, ou conditions :

— **Si** une certaine condition est vraie, exécuter un certain groupe (ou bloc) d'instructions



— **Sinon**, passer directement à la suite du programme

La syntaxe d'une instruction conditionnelle est :

```
# début
```

```
if condition:
    # bloc
    # d'instructions
    # indenté
# suite (*)
```

— Ici `condition` est une expression booléenne
— Les instructions du bloc **v** sont exécutées uniquement si `condition` est évaluée à `True`
— Dans tous les cas, l'exécution reprend à l'instruction suivant le bloc indenté (ligne `suite (*)`)

Exemple : pile ou face ? Un programme qui : 1. tire un nombre au hasard entre `0` et `1` 2. affiche `pile` s'il a tiré `1`, `face` s'il a tiré `0`

```
from random import randint # randint permet de tirer un nombre
    aléatoire

tirage = randint(0,1)
print(tirage)
if tirage == 1:
    print("pile")

if tirage == 0:
    print("face")
```

Exemple : mettre deux chaînes dans l'ordre Supposons qu'il existe deux variables `a` et `b` désignant des `str`.

Écrire un bout de programme qui modifie ces variables pour que `a` désigne la plus petite chaîne et `b` la plus grande (dans l'ordre lexicographique)

```
a = input()
b = input()
if a > b:
    # on intervertit les valeurs
    temp = a  # variable temporaire
    a = b
    b = temp
print(a, b)
```

```
a = input()
b = input()
if a > b:
    # variante
    a, b = b, a
print(a, b)
```

La notion de bloc

Sur cet exemple, on a vu un groupe de lignes commençant par des espaces, appelé **bloc**. Un bloc est utilisé pour regrouper plusieurs instructions dépendant de la même condition.

- Un tel groupe d'instructions est appelé un **bloc**
- Le décalage du début de ligne est appelé **indentation**
- Un bloc se termine quand une ligne **moins indentée** apparaît (sur l'exemple : `print(a, b)`)
- Pour indenter la ligne courante : touche "tabulation" (⇥)
- Pour désindenter une ligne : "Shift + tabulation" (⇧ + ⇥)
- Changer l'indentation change le sens du programme (essayez !)

```
1  from random import randint # randint permet de tirer un nombre
       aléatoire
2
3  tirage = randint(0,1)
4  print(tirage)
5  if tirage == 1:
6      print("Le résultat est :")
7      print("pile")
8
9  if tirage == 0:
10     print("Le résultat est :")
11     print("face")
```

Erreurs fréquentes liées à l'indentation :

```
1  if a > b  # oubli des deux points (:)
2      temp = a
3      a = b
4      b = temp
```

```
1  if a > b:
2      temp = a
3      a = b
4     b = temp  # ligne pas assez indentée
```

```
1  if a > b:
2      temp = a
3      a = b
4       b = temp  # ligne trop indentée
```

```
1  if a > b:
2  a, b = b, a  # oubli d'indentation
```

Conditionnelles `Si ... Sinon ...`

On peut ajouter un second bloc d'instructions - **Si** une certaine condition est vraie, exécuter le premier bloc - **Sinon**, exécuter le second - Enfin, continuer l'exécution normale du programme

Syntaxe :

```
# début
if condition:
    # bloc v
else:
    # bloc f
# suite
```

Ici `condition` est une expression booléenne

Seul *l'un des deux* blocs d'instructions, **v** *ou bien* **f**, est exécuté - Le bloc **v** uniquement si `condition` est évaluée à `True` - Le bloc **f** uniquement si `condition` est évaluée à `False` - Dans tous les cas, reprise à l'instruction suivant le bloc **f**

Exemple : division euclidienne

```
dividende = int(input("Donnez moi un dividende : "))
diviseur = int(input("Donnez moi un diviseur : "))
if diviseur != 0:
    quotient = dividende // diviseur
    reste = dividende % diviseur
    print(dividende, '=', quotient, '*', diviseur, '+', reste)
else:
    print('Erreur : division par zéro')
```

Exemple : pile ou face ?

```
from random import randint # randint permet de tirer un nombre
   aléatoire

tirage = randint(0,1)
print(tirage)
if tirage == 1:
    print("pile")

if tirage == 0:
    print("face")
```

```
from random import randint # randint permet de tirer un nombre
   aléatoire

tirage = randint(0,1)
```

```
print(tirage)
if tirage == 1:
    print("pile")
else: # Dans ce cas, tirage vaut nécessairement 0
    print("face")
```

Exercice

Écrire un programme qui permet de jouer à pile ou face : 1. Demander à l'utilisateur de saisir 1 pour `pile` et 0 pour face 2. Tirer un nombre au hasard, l'afficher et afficher `Gagné !` ou `Perdu !` en fonction du résultat

On pourra améliorer le programme pour permettre de saisir directement `pile` ou `face` plutôt que 1 ou 0.

```
from random import randint
nombre_choisi = int(input("Pile (1) ou face (0) ? "))
tirage = randint(0,1)
if tirage == nombre_choisi:
    print("Gagné !")
else:
    print("Perdu !")
```

```
from random import randint
choix = input("pile ou face ? ")
if choix == "pile":
    nombre_choisi = 1
else:
    nombre_choisi = 0
tirage = randint(0,1)
if tirage == 1:
    print("Le résultat est pile.")
else:
    print("Le résultat est face.")
if tirage == nombre_choisi:
    print("Gagné !")
else:
    print("Perdu !")
```

Exercices

1. Écrire un programme qui saisit un nombre entier et affiche `positif` si l'entier est positif ou nul et `negatif` sinon.
2. Écrire un programme qui saisit deux nombres entiers et affiche le plus grand des deux.

```
entier = int(input("Donnez moi un entier : "))
if entier >= 0:
    print("positif")
```

```
else:
    print("negatif")
```

```
entier1 = int(input("Donnez moi un premier entier : "))
entier2 = int(input("Donnez moi un deuxième entier : "))
print("L'entier le plus grand est", end = " ")
if entier1 > entier2:
    print(entier1)
else:
    print(entier2)
```

Conditionnelles composées

Cette construction peut être imbriquée :

```
# début
if <condition 1>:
    if <condition 2>:
        # bloc v1v2
    else:
        # bloc v1f2
    # suite 2
else:
    # bloc f1
# suite 1
```

Toutes les variantes sont possibles — si chaque **else** correspond à un **if** de même indentation !

Exemple : pile ou face, deux fois Écrire un programme qui tire deux fois à pile ou face et affiche Gagné si les deux tirages sont `pile`, `Perdu` sinon.

```
from random import randint
tirage1 = randint(0,1)
print("Premier tirage :", tirage1)
if tirage1 == 1:
    tirage2 = randint(0,1)
    print("Second tirage :", tirage2)
    if tirage2 == 1:
        print("Gagné")
    else:
        print("Perdu")
else:
    print("Perdu")
```

Exemple : discriminant On suppose qu'il existe trois variables a, b et c désignant des nombres. On veut déterminer le nombre de solutions réelles de l'équation $ax^2 + bx + c =$

0. Les cas suivants sont à considérer :

- si a = b = c = 0, il y a une infinité de solutions
- si a = b = 0 et c $\neq$ 0, il n'y a pas de solution
- si a = 0 et b $\neq$ 0, il y a exactement une solution
- sinon, on calcule le discriminant Δ = b^2 - 4ac et,
  - si $\Delta < 0$, il n'y a pas de solution ;
  - si $\Delta = 0$, il y a exactement une solution ;
  - sinon $\Delta > 0$ et il y a exactement deux solutions.

Écrire un programme qui demande à l'utilisateur de saisir au clavier les trois valeurs a, b et c et qui calcule et affiche le nombre de solutions **réelles** de l'équation du second degré associée.

```
a, b, c = ...

# Calcul et affichage du nombre de solutions
if a == 0:
    if b == 0:
        if c == 0:
            print("Une infinité de solutions")
        else:
            print("Pas de solution")
    else :
        print("Une solution")
else:
    delta = b ** 2 - 4 * a * c
    if delta < 0:
        print("Pas de solution")
    else:
        if delta == 0:
            print("Une solution")
        else:
            print("Deux solutions")
```

Conditionnelles enchaînées

Cas particulier où le bloc **else** contient seulement un autre **if** : le mot-clé elif

Le code…

```
# début
if <condition 1>:
    # bloc **v1**
else:
    if <condition 2>:
        # bloc **f1v2**
    else:
        # bloc **f1f2**
```

```
# suite
```

… s'écrit aussi :

```
# début
if <condition 1>:
    # bloc **v1**
elif <condition 2>:
    # bloc **f1v2**
else:
    # bloc **f1f2**
# suite
```

On peut ainsi enchaîner autant de conditions qu'on le souhaite, lorsque les cas ne se recouvrent pas :

```
# début
if <condition 1>:
    # bloc **v1**
elif <condition 2>:
    # bloc **f1v2**
elif <condition 3>:
    # bloc **f1f2v3**
else:
    # bloc **f1f2f3**
# suite
```

Exemple : pile ou face, deux fois (variante) Écrire un programme qui tire deux fois à pile ou face et affiche `Gagné` si les deux tirages sont différents (`pile` puis `face` ou bien `face` puis `pile`) et affiche `Perdu` sinon.

```
from random import randint
tirage1 = randint(0,1)
print("Premier tirage :", tirage1)
tirage2 = randint(0,1)
print("Second tirage :", tirage2)
if tirage1 == 1 and tirage2 == 1:
    print("Gagné")
elif tirage1 == 0 and tirage2 == 0:
    print("Gagné")
else:
    print("Perdu")
```

```
# Une variante
from random import randint
tirage1 = randint(0,1)
print("Premier tirage :", tirage1)
tirage2 = randint(0,1)
print("Second tirage :", tirage2)
```

```
if tirage1 == 1 and tirage2 == 1 or tirage1 == 0 and tirage2 ==
   0:
    print("Gagné")
else:
    print("Perdu")
```

```
# Une autre variante
from random import randint
tirage1 = randint(0,1)
print("Premier tirage :", tirage1)
tirage2 = randint(0,1)
print("Second tirage :", tirage2)
if tirage1 == tirage2:
    print("Gagné")
else:
    print("Perdu")
```

Boucles

Comment faire si l'on veut répéter une instruction ?

Exemple : pile ou face, rejouer tant qu'on perd

Écrire un programme qui nous fait rejouer à pile ou face tant qu'on perd.

```
# Un premier essai
from random import randint
tirage = randint(0,1)
nombre_choisi = int(input("Pile (1) ou face (0) ? "))
if tirage == nombre_choisi:
    print("Gagné")
else:
    print("Perdu. Essaye encore.")
    tirage = randint(0,1)
    nombre_choisi = int(input("Pile (1) ou face (0) ? "))
    if tirage == nombre_choisi:
        print("Gagné")
    # ...
    # Ça peut durer longtemps !
```

```
# Un deuxième essai
from random import randint
gagne = False
while not(gagne):
    tirage = randint(0,1)
    nombre_choisi = int(input("Pile (1) ou face (0) ? "))
    if tirage == nombre_choisi:
        print("Gagné")
```

```
        gagne = True
    else:
        print("Perdu. Essaye encore.")
```

```
# Une variante
from random import randint
tirage = 0
nombre_choisi = 1
while tirage != nombre_choisi:
    tirage = randint(0,1)
    nombre_choisi = int(input("Pile (1) ou face (0) ? "))
    if tirage != nombre_choisi:
        print("Perdu. Essaye encore.")
print("Gagné !")
```

Exemple : pile ou face, rejouer

Écrire un programme qui nous fait rejouer à pile ou face tant qu'on perd et qu'on veut rejouer.

```
from random import randint
rejouer = True
perdu = True
while rejouer and perdu:
    tirage = randint(0,1)
    nombre_choisi = int(input("Pile (1) ou face (0) ? "))
    if tirage == nombre_choisi:
        perdu = False
    else:
        print("Perdu.")
        reponse = input("Voulez-vous rejouer ? [o/n]")
        if reponse == "n":
            rejouer = False
if not perdu:
    print("Gagné")
```

Exemple : Compter de 1 à 100

```
# à corriger !
i = 0
while i < 100:
    print(i+1, end=" ")
    i = i + 1  # ou bien : i += 1
```

Exercice

1. Compter de 1 à 100 par pas de 2, de 3…
2. Compter 100 à 1 par pas de -1, de -2…
3. Compter de a à b par pas de c pour a, b et c trois entiers quelconques. Dans quels cas a-t-on des problèmes ?

```
i = 1
pas = 4
while i <= 100:
    print(i, end=" ")
    i = i + pas  # ou bien : i += 1
```

```
i = 100
pas = -4
while i > 0:
    print(i, end=" ")
    i = i + pas  # ou bien : i += 1
```

```
a, b, c = 27, 42, 2
i = a
pas = c
while i <= b:
    print(i, end=" ")
    i = i + pas  # ou bien : i += 1
```

Exercice

1. Simuler le lancer de 10 000 pièces et calculer la proportion de “pile” obtenue
2. Simuler le lancer d’une pièce jusqu’à la première “face” obtenue, et afficher le nombre de lancers effectués

```
nb_lancers = 1e6

i = 1
pile = 0
while i <= nb_lancers :
    lancer = randint(0, 1)
    if lancer == 0:
        pile += 1
    i += 1
print(pile/nb_lancers)
```

Exemple : Jouer à pierre-feuille-ciseau

```
from random import randint

coup_humain = int(input("Pierre (1), feuille (2) ou ciseaux (3)
   ? "))
```

```
4
5  coup_ordi = randint(1, 3)
6  if coup_ordi == 1:
7      print("L'ordinateur a joué pierre.")
8  elif coup_ordi == 2:
9      print("L'ordinateur a joué feuille.")
10 else:
11     print("L'ordinateur a joué ciseaux.")
12
13 if coup_ordi == coup_humain:
14     print("Égalité.")
15 elif coup_humain == (coup_ordi + 1) % 3:
16     print("Vous avez gagné.")
17 else:
18     print("Vous avez perdu.")
```

Exercice

1. Expliquer la ligne 15
2. Expliquer ce qu'il se passe si l'humain entre 4.
3. Proposer de rejouer une partie
4. Afficher le nombre de parties jouées et le score final

Répétition simple

On peut répéter un bloc d'instructions grâce à une boucle « tant que » ou boucle **`while`**.

— **Si** une certaine condition est vraie, on va exécuter un certain bloc d'instructions ;
— **Sinon**, on va passer directement à la suite du programme ;
— Après chaque exécution du bloc, on réévalue la condition.

Vocabulaire :

— L'expression booléenne `condition` est appelée **condition de continuation**.
— Sa négation (`not condition`) est appelée **condition d'arrêt**.
— Le bloc d'instructions est appelé **corps de la boucle**.
— Chaque exécution du corps de la boucle est appelée **itération**.

Syntaxe :

```
1 # début
2 while condition:
3     # bloc d'instructions
4     # (corps de la boucle)
5 # suite
```

— `condition` est une **expression booléenne**
— corps exécuté uniquement si `condition` s'évalue à `True`

- après chaque exécution du corps, on réévalue `condition`
  - si `condition` s'évalue à `False`, sortie de la boucle
  - sinon, nouvelle **itération**

Il peut n'y avoir aucune itération, ou un nombre infini !

Outil d'analyse : tableau de valeurs

Utile pour exécuter manuellement une boucle

- Une colonne pour indiquer le numéro de la dernière ligne du programme exécutée
- Une colonne pour indiquer le nombre d'itérations exécutées
- Une colonne par variable "intéressante"
- Une colonne pour la condition de continuation
- Éventuellement des colonnes explicatives supplémentaires

On remplit le tableau au moins pour **la ligne précédant la boucle** et pour **la dernière ligne du corps**

Exemple : Calculer a^p

On veut calculer 2^n (sans utiliser `**`).

Algorithme naïf : on remarque que $2^n = 2^{n-1} \times 2 = 1 \times 2 \times 2 \times \cdots \times 2$

1. on commence par fixer le résultat à 1
2. on multiplie le résultat par 2 `n` fois

```python
a = 2
p = 4
res = 1  # pourquoi ?
i = 0
while i < p:
    res = res * a
    i = i + 1
print(a, "puissance", p, "égale", res)
```

```
2 puissance 4 égale 16
```

ligne	itération	`a`	`p`	`res`	`i`	`i < p`	commentaire
1		2					
2		-	4				
3		-	-	1			

ligne	itération	`a`	`p`	`res`	`i`	`i < p`	commentaire
4		-	-	-	0	`True`	
5		-	-	-	-	`True`	condition vraie, on entre
6	1	-	-	2	-	`True`	
7	1	-	-	-	1	`True`	fin de la 1e itération
5		-	-	-	-	`True`	condition vraie, on continue
6	2	-	-	4	-	`True`	
7	2	-	-	-	2	`True`	fin de la 2e itération
5		-	-	-	-	`True`	condition vraie, on continue
6	3	-	-	8	-	`True`	
7	3	-	-	-	3	`True`	fin de la 3e itération
5		-	-	-	-	`True`	condition vraie, on continue
6	4	-	-	16	-	`True`	
7	4	-	-	-	4	`False`	fin de la 4e itération
5		-	-	-	-	`False`	condition fausse, on arrête
8		-	-	-	-	`False`	suite du programme

Remarques : - après la ligne 4 et chaque exécution de la ligne 7 on a `res == a**i` - `i` se rapproche de `p` à chaque tour sans le dépasser

Version “compacte” sans regarder toutes les lignes ni les variables qui ne changent pas :

ligne	itération	`res`	`i`	`i < p`	commentaire
4		1	0	`True`	juste avant la première itération
7	1	2	1	`True`	à la fin de la 1e itération
7	2	4	2	`True`	à la fin de la 2e itération
7	3	8	3	`True`	à la fin de la 3e itération
7	4	16	4	`False`	à la fin de la 4e itération (sortie)

Remarques : - pendant tout le programme `a` vaut 2 et `p` vaut 4 - après la ligne 4 et chaque

exécution de la ligne 7 on a `res == a**i` - `i` se rapproche de `p` à chaque tour sans le dépasser - à la fin de la dernière itération `i == p` et donc `res == a**p`

Variante

```
a = 2
p = 4
res = 1
i = p  # changement !
while i > 0:  # changement !
    res *= a
    i -= 1  # changement !
print(a, "puissance", p, "égale", res)
```

Tableau de valeurs compact :

ligne	itération	res	i	i > 0	commentaire
4		1	4	`True`	avant la première itération
7	1	2	3	`True`	
7	2	4	2	`True`	
7	3	8	1	`True`	
7	4	16	0	`False`	sortie de la boucle

Remarques : - pendant tout le programme `a` vaut 2 et `p` vaut 4 - après la ligne 4 et chaque exécution de la ligne 7 on a `res == a**(p-i)` - la valeur de `i` est positive au début et décroît strictement - à la fin de la dernière itération `i == 0` et donc `res == a**(p-0) == a**p`

Terminaison et correction d'une boucle

En général rien ne garantit :

— qu'une boucle **`while`** va se terminer un jour

```
while(True):
    print("spam")
```

— ni qu'elle produit le bon effet

Pour cela il faut en général faire des **preuves**

Les deux sections suivantes présentent des méthodes classiques pour présenter ces preuves : elles ne sont pas exigibles au contrôle. Néanmoins intéressantes et importantes, elles seront d'ailleurs détaillées dans le cours *Algorithmique et structures de données* en L2.

Preuve de terminaison : variant Méthode possible pour montrer qu'une boucle termine

- montrer qu'une certaine quantité décroît strictement à chaque tour de boucle
- montrer qu'elle ne peut pas décroître indéfiniment

On appelle une telle quantité **variant de boucle**, son existence garantit la terminaison

Exemple : algorithme d'Euclide Algorithme de l'antiquité permettant de déterminer le PGCD de deux nombres entiers

```
1 a0, b0 = 129, 36  # entiers positifs quelconques
2
3 a, b = a0, b0
4 while b > 0:
5     r = a % b
6     a = b
7     b = r
8
9 print("le pgcd de", a0, "et", b0, "est", a)
```

Pourquoi l'algorithme termine-t-il ?

- on peut choisir comme variant la **valeur de b**
- initialement, `b > 0`
- la boucle ne s'exécute pas si `b <= 0`
- la valeur de `b` décroît strictement

Comme il ne peut exister de suite infinie strictement décroissante d'entiers positifs, la boucle termine

On peut repérer le variant dans le **tableau de valeurs**

ligne	itération	a	b	a % b	commentaire
3		129	36	21	avant la boucle
7	1	36	21	15	fin de 1e itération
7	2	21	15	6	…
7	3	16	6	3	
7	4	6	3	0	
7	5	3	0	-	on va sortir de la boucle

Variant : la valeur de `b` est positive au début et décroît strictement

Preuve de correction : invariant Méthode possible pour montrer qu'une boucle produit le bon effet :

— montrer qu'une certaine propriété I est vraie avant l'entrée dans la boucle
— montrer que **si** I est vraie au début du corps **alors** elle est encore vraie à la fin
— en déduire que I est vraie à la sortie de la boucle

On appelle une telle propriété **invariant**, son existence peut permettre de garantir la correction

Exemple : retour sur le calcul de puissance

```
1 a = 2
2 p = 4
3 res = 1
4 i = 0
5 while i < p:
6     res *= a
7     i += 1
8 print(a, "puissance", p, "égale", res)
```

ligne	itération	`res`	`i`	`i < p`	commentaire
4		1	0	`True`	juste avant la première itération
7	1	2	1	`True`	à la fin de la 1e itération
7	2	4	2	`True`	à la fin de la 2e itération
7	3	8	3	`True`	à la fin de la 3e itération
7	4	16	4	`False`	à la fin de la 4e itération (sortie)

Remarques : - **Variant :** `p - i` décroît de 1 à chaque tour et la boucle s'arrête quand il atteint `0` - **Invariant :** après la ligne 4 et chaque exécution de la ligne 7 on a `res == a**i` - **En sortie de boucle :** `i == p` et donc `res == a**p`

Variante

```
1 a = 2
2 p = 4
3 res = 1
4 i = p
5 while i > 0:
6     res *= a
7     i -= 1
8 print(a, "puissance", p, "égale", res)
```

ligne	itération	`res`	`i`	`i > 0`	commentaire
4		1	4	`True`	avant la première itération
7	1	2	3	`True`	
7	2	4	2	`True`	
7	3	8	1	`True`	
7	4	16	0	`False`	sortie de la boucle

Remarques : - **Variant :** `i` décroît de 1 à chaque tour et la boucle s'arrête quand il atteint 0 - **Invariant :** après la ligne 4 et chaque exécution de la ligne 7 on a `res == a**(p-i)` - **En sortie de boucle :** `i == 0` et donc `res == a**(p-0)== a**p`

Exemple : encadrer un nombre On veut encadrer un nombre positif n entre deux puissances successives d'un nombre b. On cherche l'unique entier k tel que :

$$b^k \leq n < b^{k+1}$$

On appelle parfois k le *logarithme entier* de n en base b, parce que

$$k \leq \log_b n < k + 1$$

Par exemple :

- pour $b = 10$ on a $10^3 \leq 1024 < 10^4$, donc $k = 3$.
- pour $b = 2$ on a $2^{10} \leq 1024 < 2^{11}$, donc $k = 10$.

```
1 n = 1000  # le nombre à encadrer
2 b = 10    # base de la puissance
3 exp = 0   # exposant courant
4 temp = 1  # valeurs successives de b**exp
5 while temp <= n:
6     temp *= b
7     exp += 1
8 print("le plus petit k tel que", b, "à la puissance k "
9       "est inférieur ou égal à", n, "est", exp-1)  # compléter
```

ligne	temp	exp	temp < n	commentaire
4	1	0	`True`	avant la première itération
7	10	1	`True`	

ligne	temp	exp	temp < n	commentaire
7	100	2	`True`	
7	1000	3	`True`	
7	10000	4	`False`	on sort de la boucle

— **Variant :** `n - temp` initialement ≥ 0, diminue à chaque tour
— **Invariant** (simplifié) lignes 4 et 7 : `temp == 10**exp` et `10**(exp-1) <= n`
— **À la fin :** `10**(exp-1) <= n` et `10**exp > n`, autrement dit `10**(exp-1) <= n < 10**exp`

Exemple : convertir un nombre en binaire Le nombre 42 s'écrit `101010` en binaire parce que

$$\begin{aligned}
42 &= 0 + 2 \times 21 \\
&= 0 + 2 \times (1 + 2 \times 10)) \\
&= \dots \\
&= 0 + 2 \times (1 + 2 \times (0 + 2 \times (1 + 2 \times (0 + 2 \times (1 + 2 \times 0))))) \\
&= 0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 1 \times 2^5
\end{aligned}$$

```
1 bin(42)
```

Algorithme de conversion de $n > 0$ en binaire :

1. Calculer le quotient q et le reste r de n par 2
2. Ajouter r comme nouveau chiffre à **gauche** du résultat
3. Poser $n = q$ et recommencer en 1 si $n > 0$

Suite de l'exemple :

$$
\begin{aligned}
42 &= 2 * 21 + 0 &&\rightarrow \text{chiffre } 0\\
21 &= 2 * 10 + 1 &&\rightarrow \text{chiffre } 1\\
10 &= 2 * 5 + 0 &&\rightarrow \text{chiffre } 0\\
5 &= 2 * 2 + 1 &&\rightarrow \text{chiffre } 1\\
2 &= 2 * 1 + 0 &&\rightarrow \text{chiffre } 0\\
1 &= 2 * 0 + 1 &&\rightarrow \text{chiffre } 1
\end{aligned}
$$

```python
n = 42
k = n
res = ""  # on utilise une chaîne
while k > 0:
    q = k // 2
    r = k % 2
    res = str(r) + res
    k = q
print(res)
```

ligne	itér	k	r	res	k × $2^{\text{itér}}$	val. res	k > 0	commentaire
3	0	42		''	$42 \times 2^0 = 42$	0	True	avant la première itération
8	1	21	0	'0'	$21 \times 2^1 = 42$	0	True	fin de la 1e itération
8	2	10	1	'10'	$10 \times 2^2 = 40$	2	True	
8	3	5	0	'010'	$5 \times 2^3 = 40$	2	True	
8	4	2	1	'1010'	$2 \times 2^4 = 32$	10	True	
8	5	1	0	'01010'	$1 \times 2^5 = 32$	10	True	
8	6	0	1	'101010'	$0 \times 2^6 = 32$	42	False	sortie de la boucle

— **Invariant** : `res` contient les (nombre d'itérations) derniers chiffres de la conversion de `n` en binaire

$k \times 2^{(\text{itér})} +$ (valeur de res) $= n$

— **Variant :** `n - temp` initialement ≥ 0, diminue à chaque tour

Exercice : - dresser le tableau de valeurs pour $n = 25$ - adapter l'algorithme pour convertir n en base 4, puis en base $b < 10$

ref : Compter comme les Shadoks

Exercice : quel est l'invariant de boucle pour l'algorithme d'Euclide ?

Boucles imbriquées

On peut écrire une boucle à l'intérieur d'une autre boucle

La syntaxe d'une double boucle **`while`** :

```
# début
while condition1:
    # début corps 1
    while condition2:
        # corps 2
    # fin corps 1
# suite
```

Exemple : table d'addition But : afficher toutes les additions de nombres inférieurs à `n`

```
m = 3
n = 11

a = 0  # hors de la boucle externe !
while a < m:
    b = 0  # dans la boucle externe !
    while b < n:
        print(a, '+', b, '=', a+b)
        b += 1  # dans la boucle interne !
    a += 1  # dans la boucle externe !
```

Exercices : 1. afficher toutes les additions `a + b = c` pour `a` entre 0 et `n` et `b` entre 0 et `m` 2. essayer d'écrire le programme en utilisant une seule boucle (*non recommandé en temps normal !*)

Exemple : compter jusqu'à 59, dix nombres par ligne

```
# paramètres
limite = 60
nb_colonnes = 7

courant = 0
```

```
while courant < limite:
    colonne = 0
    while colonne < nb_colonnes and courant < limite:
        print(courant, end = '\t')
        colonne += 1
        courant += 1
    print()  # affiche un retour à la ligne
```

Exercice : (pas si évident) faire en sorte que les nombres successifs apparaissent sur une même colonne

```
# paramètres
limite = 37
nb_colonnes = 10

nb_lignes = limite // nb_colonnes + 1
ligne = 0
while ligne < nb_lignes:
    colonne = 0
    courant = ligne
    while courant < limite and colonne < nb_colonnes:
        print(courant, end = '\t')
        courant += nb_lignes
        colonne += 1
    ligne += 1
    print()  # affiche un retour à la ligne
```

Contrôle de boucle

Tout ce qui suit est *non exigible en contrôle*.

Instruction `break` On peut sortir prématurément d'une boucle **`while`** - Instruction **`break`** dans le corps (en général dans une conditionnelle) - Force une sortie de boucle et passe directement à la suite du programme - On ne réévalue pas la condition du **`while`**

Exemple : saisie contrôlée sans duplication de l'instruction `input` :

```
while True:
    note_cc1 = float(input('Note du premier contrôle : '))
    if 0 <= note_cc1 <= 20:
        # saisie correcte, on termine la boucle
        break
    # saisie incorrecte, on recommence
    print('Erreur de saisie.')
```

Instruction `continue` On peut passer prématurément à l'itération suivante - Instruction `continue` dans le corps de la boucle - On retourne directement à la condition, et on la réévalue

```
while True:
    saisie = input("Dis-moi quelque chose : ")
    if saisie == "":
        print("Tu n'as rien dit !")
        continue
    if saisie == "je réfléchis":  # elif inutile ici !
        print("OK, j'attends...")
        continue
    if saisie == "stop":  # elif inutile ici !
        print("D'accord...")
        break
    if saisie == "arrête":  # elif inutile ici !
        print("C'est bon, j'ai compris !")
        break
    # else inutile ici !
    print("Ta phrase fait", len(saisie), "caractères.")
```

Remarques sur `break` et `continue`

- Ces deux instructions ne sont à utiliser que par des programmeurs confirmés
- Elles compliquent en général la compréhension et l'analyse du code
- On peut toujours s'en sortir sans (en utilisant des `if`), même si ça peut rendre le code moins "élégant"

```
saisie = ""
while saisie != "stop" and saisie != "arrête":
    saisie = input("Dis-moi quelque chose : ")
    if saisie == "":
        print("Tu n'as rien dit !")
    elif saisie == "je réfléchis":
        print("OK, j'attends...")
    else:
        print("Ta phrase fait", len(saisie), "caractères.")
if saisie == "stop":
    print("D'accord...")
elif saisie == "arrête":
    print("C'est bon, j'ai compris !")
```

```
a, b, c = ...

# Calcul et affichage du nombre de solutions
if a == b == c == 0:
    print("Une infinité de solutions réelles (et complexes)")
elif a == b == 0:  # A ce stade, si a = b = 0, c est différent
   de 0 !!!
    print("Aucune solution")
```

```
8  elif a == 0 :      # A ce stade, si a = 0, b est différent de 0
       !!!
9      print("Une solution réelle")
10 else:
11     delta = b ** 2 - 4 * a * c
12     if delta < 0:
13         print("Pas de solution réelle")
14     elif delta == 0:
15         print("Une solution réelle")
16     else:
17         print("Deux solutions réelles")
```

Introduction aux listes

Il arrive souvent que l'on veuille faire référence à plusieurs données en même temps (parce qu'ils sont de même type, ou parce qu'elles se rapportent au même objet, par exemple). Par ailleurs, on ne sait pas toujours quelle quantité de données on va devoir retenir pour le bon déroulé de notre programme ou algorithme. Difficile dans ces cas de savoir le nombre exact de variables à déclarer et initialiser ! On a donc envie de pouvoir regrouper des données en les organisant en *collections ordonnées*, ayant en plus un caractère *dynamique* et *mutable* : on peut ajouter des éléments à notre collection à notre guise, en supprimer, en modifier…

En algorithmique, on parle souvent de *tableaux*, en Python, le type correspondant est **list**, et on parlera sans arrêt de *listes*.

Nous approfondirons tout cela dans la séquence de cours suivante (on verra en particulier en détail les notions de *mutabilité*, d'*itérabilité*, de boucle * for*), mais voici déjà quelques bases pour manipuler ces objets un peu spéciaux mais très utiles !

Pour résumer :

Objectif : désigner avec une seule variable une collection de valeurs

Liste : suite **indexée** (numérotée) d'objets quelconques (type `list` en python)

- Élements "rangés" dans des "cases" numérotées de 0 à $n-1$
- En mémoire : tableau à n cases, chacune contenant une référence (*"flèche"*) vers un objet
- Peut contenir des objets de plusieurs types différents
- **Mutable** : peut être modifiée, agrandie, raccourcie…

Une métaphore

Un peu plus proche de la réalité

Création et affichage

Création : suite entre [et] d'expressions séparées par ,

```
lst = [3, 'toto', 4.5, False, None]
print(lst)
```

Liste vide [] : liste ne contenant aucun objet

```
lst = []
print(lst)
```

Une liste peut contenir d'autres listes !

```
lst = ['test', [1, [2], 3]]
print(lst)
```

[Exemple (Python tutor)](http ://pythontutor.com/visualize.html#code=%23Cr%C3%A9ation%20et%20affi
ce%20que%20lst_ex2%20est%20modifi%C3%A9e%3F%5Cn%22,%20lst_ex2%29%0A%0Alst_ex2%5B2%5
ce%20que%20val_ex2%20est%20modifi%C3%A9e%3F%5Cn%22,%20val_ex2%29%0A%0A%23ajout%20
frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false)
(exemple filé à exploiter tout au long du cours)

Opérations et fonctions de base

Longueur d'une liste La longueur d'une liste (le nombre d'éléments qu'elle contient) s'obtient par la fonction `len`.

```
lst = [3, 'toto', 4.5, False, None]
print(len(lst))
```

```
print(len([]))
```

Accès aux éléments Les éléments d'une liste à n éléments sont numérotés de 0 à $n-1$.
Le numéro d'un élément est appelé son *indice*.
L'accès à un élément donné s'appelle l'**indexation**.

```
lst = [3, 'toto', 4.5]
print(lst[1])
```

Attention !

- Le premier élément d'une liste est l'élément d'indice `0` !
- Si la liste a `n` éléments, il n'existe pas d'élément d'indice `n` !

— L'accès à un indice supérieur ou égal à la taille de la liste provoque une erreur !

```
lst = [3, 'toto', 4.5]
print(lst[3])
```

Exercice : Écrire une fonction qui affiche tous les éléments d'une liste (un par ligne)

```
def affiche_elements(lst):
    ...

lst = [3, 'toto', 4.5]
affiche_elements(lst)
```

```
def affiche_elements(lst):
    i = 0
    while i < len(lst):
        print(lst[i])
        i = i + 1


lst = [3, 'toto', 4.5]
affiche_elements(lst)
```

Modification d'un élément On peut modifier le i-ème élément de `lst` à l'aide d'une affectation :

```
lst = [3, 'toto', 4.5, False, None]
print(lst[2])
lst[2] = 'titi'
print(lst)
```

Attention, ceci ne crée pas une nouvelle liste mais modifie la liste sur place !

```
lst = [3, 'toto', 4.5, False, None]
lst_bis = lst
lst[2] = 'titi'
lst_bis
```

Concaténation et répétition Comme pour les chaînes de caractères (`str`) on peut utiliser les opérateurs + pour fabriquer la concaténation de deux listes et $\star$ pour répéter une liste.

```
[3, 'toto', 4.5] + [False, None]
```

```
[] + [3, 'toto', 4.5] + []
```

```
3 * ['a', 'b']
```

```
[0] * 13
```

On peut utiliser ces opérateurs pour recopier une liste. Comparer :

```
lst = [3, 'toto', 4.5]
lst2 = lst
lst3 = lst + []
lst4 = lst * 1
```

Test d'appartenance **Exercice :** Écrire une fonction recevant une liste et une valeur, et renvoyant `True` si la valeur apparaît dans la liste (`False` sinon)

```
def appartient(lst, val):
    ...
```

```
def appartient(lst, val):
    i = 0
    while i < len(lst):
        if lst[i] == val:
            return True
        i += 1
    return False

lst = ['Hildegarde', 'Cunégonde', 'Médor']

print(appartient(lst, 'Cunégonde'))

if appartient(lst, 'Médor'):
    print('Bon chien !')
```

Remarque : Cette fonctionnalité existe déjà en Python :

— `val in lst` vaut `True` si `val` apparaît dans `lst`, `False` sinon
— Réciproquement, on peut écrire `val not in lst`

```
lst = ['Hildegarde', 'Cunégonde', 'Médor']
'Cunégonde' in lst
```

```
lst = ['Hildegarde', 'Cunégonde', 'Médor']
'Rex' not in lst
```

```
lst = ['Hildegarde', 'Cunégonde', 'Médor']
if 'Médor' in lst:
    print('Bon chien !')
```

Manipulations par méthodes

On va maintenant énumérer un certain nombre de méthodes prédéfinies sur les listes, permettant des modifications plus complexes. Pour plus de détails, on pourra consulter la docu-

mentation en ligne.

Agrandir ou rétrécir une liste Plusieurs instructions ont un effet sur la taille de la liste :

- L'instruction `lst.append(elem)` ajoute l'élément `elem` à la fin de la liste `lst`
- L'instruction `lst.pop()` supprime le dernier élément de `lst` et renvoie sa valeur
- L'instruction `lst.pop(i)` supprime l'élément d'indice `i` de `lst` et renvoie sa valeur

*Les fonctions `append` et `pop` sont appelées **méthodes**, ou fonctions s'appliquant à un objet (nous en verrons d'autres dans les cours suivants)*

Attention, ces instructions ne créent pas une nouvelle liste mais modifient la liste sur place !

Attention, ne pas confondre `x = lst[2]` et `x = lst.pop(2)` !

```
lst = [3, 'toto', 4.5, False, None]
lst_bis = lst

lst.append(1)
print(lst_bis)

elem = lst_bis.pop(2)
print(elem)

print(lst)
```

Introduction aux fonctions

En programmation, une fonction est : - un morceau de programme - portant en général un **nom** - acceptant zéro, un ou plusieurs **paramètres** - produisant le plus souvent un **résultat**.

Des exceptions existent, mais la forme la plus courante d'une fonction est donc proche de celle d'une fonction mathématique.

L'utilisation de fonctions améliore les aspects suivants du code :

- **Lisibilité :**
  - isoler une partie du programme (par exemple un gros calcul compliqué)
  - éviter une trop grande imbrication des `if`, des `while`
- **Modularité et robustesse :**
  - réutiliser le même code plusieurs fois (évite de recopier le code)
  - faciliter la correction des bugs, l'évolution et la maintenance
- **Généricité :**
  - changer la valeur des paramètres (même calcul mais avec différentes valeurs de départ)

Fonctions prédéfinies et bibliothèque standard

En Python, il existe un grand nombre de fonctions prédéfinies, que nous avons déjà utilisées, par exemple :

— `int(obj)` : 1 paramètre, 1 résultat. Reçoit en paramètre un objet (par exemple `str` ou `float`), essaie de le transformer en entier et renvoie l'entier obtenu.

```
int("34")
```

— `len(obj)` : 1 paramètre, 1 résultat. Reçoit un objet (par exemple `str`) et renvoie sa longueur.

```
len("bonjour")
```

— `randint(mini, maxi)` : 2 paramètres, 1 résultat. Reçoit deux nombres, et renvoie un entier aléatoire compris entre ces deux nombres (inclus).

```
from random import randint
randint(1, 34)
```

Il y en a beaucoup d'autres, comme `print`, `input`, `float`, `str`...

Ces fonctions sont appelées *prédéfinies* (ou *built-in*) - il est inutile de les connaître toutes par cœur - liste des fonctions prédéfinies sur cette page

Il existe également de nombreux *modules* officiels (par exemple le module `random`) - bibliothèques de fonctions, de types et d'objets - liste des modules prédéfinis documentée ici.

Définition de fonction

Pour définir une nouvelle fonction on utilise la syntaxe suivante :

```
# ligne suivante : en-tête de fonction
def nom_fonction(param_1, ..., param_n):
    # corps de la fonction
    # utilisant param_1 à param_n
    ...
    # peut renvoyer un résultat :
    return resultat
```

Une fonction peut :

— prendre un certain nombre de paramètres (ici, n, qui s'appellent `param_1` à `param_n`)
— renvoyer une valeur (via l'instruction `return`)

Appel de fonction

Une fois définie, `nom_fonction` peut être utilisée dans le code (on parle d'un **appel**) en indiquant entre parenthèses ses paramètres séparés par des virgules :

```
# définition de fonction
def nom_fonction(param_1, ..., param_n):
    ...

# reste du programme
...
# appel de la fonction :
une_var = nom_fonction(expr_1, ..., expr_n)
```

Exemples

Fonction à paramètres et résultat

```
# fonction à deux paramètres produisant un résultat
def maximum(a, b):
    if a >= b:
        return a
    else:
        return b
```

```
nb1 = 14
nb2 = 31

maximum(nb1, nb2)  # ne sert à rien !!

# On appelle la fonction et on garde le résultat dans c :
c = maximum(nb1, nb2)
print("le max de", nb1, "et", nb2, "est", c)

# On peut aussi utiliser directement le résultat :
print("le max de", nb1, "et", nb2, "est", maximum(nb1, nb2))
```

Entraînement :

— Décrire l'exécution pas à pas du programme (avec état de la mémoire). On peut aussi essayer avec Python Tutor.
— Dresser un tableau de valeurs de l'exécution du programme

Fonction sans paramètre

— En principe, une fonction sans paramètre devrait avoir toujours le même comportement
— Dans l'exemple suivant, on utilise un générateur pseudo-aléatoire, ce qui explique que la fonction ne renvoie pas toujours le même résultat

— Une fonction pourrait aussi recevoir des données depuis l'extérieur (utilisateur, requête réseau…).

Dans ces cas, on parle de **causes secondaires**

```
from random import randint

def lance_de() :
    return randint(1,6)

compteur = 1
while lance_de() != 6:
    compteur = compteur + 1
print('Obtenu un 6 en', compteur, 'jets de dé.')
```

Fonction sans valeur de retour

— Si l'exécution arrive à la dernière instruction du corps de la fonction sans rencontrer d'instruction `return expr`, alors la valeur de retour par défaut est `None` (même comportement si `return` seul)

— En général une telle fonction a quand même un effet sur l'environnement (affichages, dessin, écriture dans un fichier, envoi d'informations sur le réseau…)

Pour tous les effets autres que le renvoi d'un résultat, on parle d'**effets secondaires**

```
from turtle import *

def trace_polygone(nb_cotes, taille_cote):
    down()
    i = 0
    while i < nb_cotes:
        forward(taille_cote)
        left(360 / nb_cotes)
        i = i + 1

# faire une affectation ici ne servirait à rien (essayer !)
trace_polygone(5, 100)
exitonclick()
```

Erreur fréquente : confusion (paramètre / saisie) et (retour / affichage)

Les programmeurs débutants confondent très souvent la notion de paramètre et celle de saisie (au clavier par exemple) et la notion de valeur de retour avec celle de valeur affichée.

```
# ATTENTION CECI EST INCORRECT, A NE PAS REPRODUIRE !!!
def pgcd(a, b):
    a = int(input())  # NON !
    b = int(input())  # NON !
    while a % b != 0:
```

```
6           r = a % b
7           a = b
8           b = r
9       print("le pgcd est", b)  # NON !
```

NE SURTOUT PAS PROGRAMMER COMME ÇA !

— Comment écrire un programme vérifiant si trois entiers sont premiers entre eux à l'aide de cette fonction ?
— Combien ce programme ferait-il de saisies ?
— Qu'afficherait ce programme ?