1ère NSI — Thème 3: Langage & Programmation

Introduction à la Programmation *(en Python)*

Lycée Fustel de Coulanges, Massy

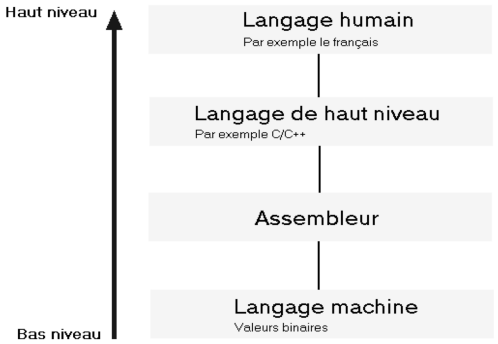
Marc Biver, septembre 2023, *v0.5*

Ce document reprend les notions abordées en cours et pratiquées en TP; il inclut la totalité de ce qui a été diffusé en classe sur ce thème. Si vous avez des questions à son propos n'hésitez pas à me contacter par le biais de la messagerie de l'ENT.

Table des matières

1 Qu'est-ce qu'un langage de programmation ?

— Programmer = demander à un ordinateur d'effectuer des actions.
— Exemple : "calcule-moi la valeur de 6 + 7".
— Problème : un ordinateur ne "parle" que le binaire ; nous, les humains, pas du tout...
— Solution : on a créé des langages dits "évolués" ou "haut niveau" qui nous permettent d'exprimer de manière naturelle ce que l'on veut que l'ordinateur réalise.



— Un langage de programmation...
  — Comporte une syntaxe, du vocabulaire, une grammaire... comme une langue humaine.
  — A l'inverse d'une langue humaine ils n'ont qu'une vocation très étroite : écrire des algorithmes de manière à ce qu'un ordinateur puisse les exécuter.
  — La traduction vers le langage machine que comprend l'ordinateur peut se faire de deux manière :
    — La compilation l'équivalent de la traduction d'un livre — cela produit un nouveau fichier, appelé "exécutable" qui peut être directement exécuté par l'ordinateur (comme un livre traduit peut être directement lu).
    — L'interprétation : comme "dans la vraie vie" c'est une traduction au fur et à mesure que le code est écrit, sans enregistrement du code traduit dans un nouveau fichier.
  — Exemples :
    — Le langage C utilise un compilateur — on dit que c'est un langage compilé.
    — Le langage Python utilise un interpréteur — on dit que c'est un langage interprété.

2 Le langage Python — premier contact

2.1 A propos de Python...

Pourquoi est-il celui que l'on étudie ici ?

- A notre niveau pour commencer :
  - Il est "open source" — gratuit et utilisable par tous.
  - Il est "portable" — utilisable sous Linux, MacOS, Windows.
  - Il a une syntaxe simple (*si ! si !*).
- Il a d'autres caractéristiques dont nous parlerons quand nous aborderons les thèmes qui y correspondent :
  - Il gère lui-même ses ressources mémoire.
  - Il est multi-paradigmes : impératif, fonctionnel, orienté objet.

Comment et où l'utiliser ?

On le comprend — taper du code dans un fichier texte ne suffit pas à créer un programme (même dans le cas du HTML qu'on a vu la semaine passée, il y avait un interpréteur — le navigateur).

On va utiliser ce qu'on appelle un IDE ("integrated development environment") — il en existe littéralement des dizaines. On peut citer, pour les plus proches de nous :

- IDLE — qui est installé sur les machines NSI et que l'on va utiliser pour découvrir la console ;
- Basthon — outil en ligne sur lequel s'appuie l'application Capytale que nous allons utiliser bientôt également ;
- EduPython — qui est installé sur les machines qui vous ont été fournies par la Région Ile-de-France ;
- Google Colab — outil en ligne que l'on utilisera sans doute plus tard dans l'année.

2.2 Python comme calculatrice

Dans ce document tous les encadrés à bordure rouge tels que celui ci-dessous dénotent une activité que nous avons réalisée en classe — et que vous devriez donc pouvoir retrouver dans votre ENT.

— Lancer IDLE
— Testez dans la console les opérations arithmétiques usuelles : `+ - * /`

```
>> 3 + 2
5
>> 9 / 5
1.8
>>
```

— Testez les opérations `//` et `%` — devinez-vous ce qu'elles font ?

2.3 Entrées & sorties ; affectations

Une **variable** est une manière de stocker une information dans la machine en la nommant.

Faire **l'affectation** d'une variable c'est en modifier la valeur — soit en la *créant* si elle n'existe pas encore, soit en la *changeant*.

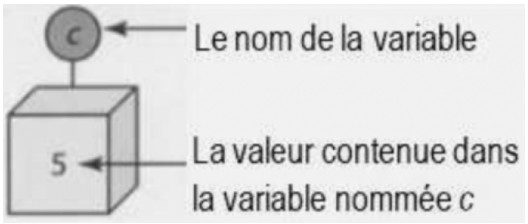
Considérons l'algorithme suivant :

```
Saisir a
Saisir b
Mettre (a + 2 x b) dans c
Mettre (c + 1) dans c
Ecrire c
```

On a ici :

- Trois variables ;
- Deux entrées ;
- Quatre affectations (deux sur des valeurs demandées à l'utilisateur, deux calculées par l'ordinateur) ;
- Une sortie, par écriture à l'écran.

Une variable est caractérisée en premier lieu par un nom (qui permet de la localiser en mémoire) et une valeur ; elle peut être vue comme une boîte étiquetée (son nom) qui peut contenir différentes informations (sa valeur) — dans le cas précédent, des nombres entiers. Affecter une variable, c'est modifier ce que contient la boîte.



Une variable a également un type — nombre entier, nombre réel, chaîne de caractères... — mais à l'inverse d'autres langages, il est implicite : il n'y a pas besoin de le déclarer.

Pour affecter une valeur à une variable en Python, on utilise le symbole =, le membre de gauche prenant la valeur de celui de droite :

```
# La ligne suivante affecte la valeur 3 a "variable"
variable = 3
```

On notera ici la syntaxe des commentaires en Python (que j'espère que vous utiliserez beaucoup !!) qui sont préfixés par un "hashtag".

ATTENTION à ne pas confondre avec les maths :

- "Variable" n'a pas le même sens ;
- Le symbole "=" n'a pas le même sens.

Pour nommer une variable, il faut respecter certaines règles :

- Caractères alphanumériques ;

— Ne pas commencer par un chiffre ;
— Aucun caractère spécial hormis l'underscore "_" ;
— On évite les accents - même si en principe ils ne sont pas interdits ;
— Et les mots clés — int, and, break... — sont interdits (on y reviendra).

— Créez plusieurs variables et affectez-y différentes valeurs, cette fois de différents types (entiers, réels, texte...). (vous pouvez utiliser les exemples ci-dessous — mais inventez-en d'autres !)

```
# Quelques exemples d'affectations pour s'entrainer...
MaVariable = 5
Fustel = 6.578
accueil = "bonjour"
```

— Affichez la valeur de ces variables au moyen de la fonction :

`print()`

— Demandez à l'utilisateur de renseigner certaines variables au moyen de la fonction input :

`age_prof = "Quel âge avez-vous? "`

— Vérifiez le type des variables que vous avez créé au moyen de la commande :

`print(type(NOM_VARIABLE))`

— On rappelle les opérations usuelles :

Instruction	Signification
`a + b`	addition
`a - b`	soustraction
`a * b`	multiplication
`a ** b`	puissance : a^b
`a / b`	division
`a // b`	division entière (quotient de la divition euclidienne)
`a % b`	modulo (reste de la divition euclidienne)

— Faites des tests d'opérations usuelles avec l'ensemble des variables que vous avez affectées précédemment — rencontrez-vous des erreurs ? Lesquelles ?

Exercices

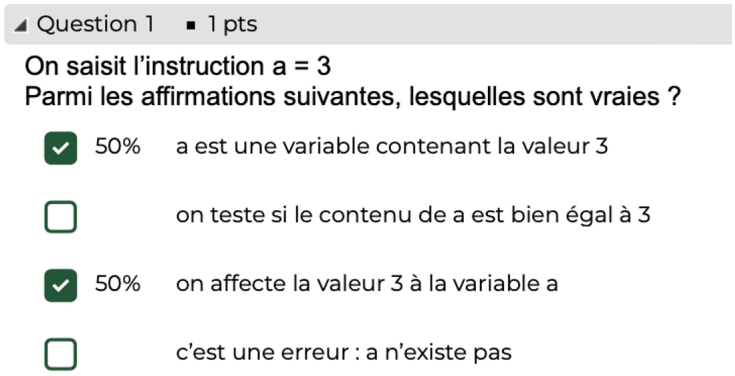
On considère le programme Python suivant :

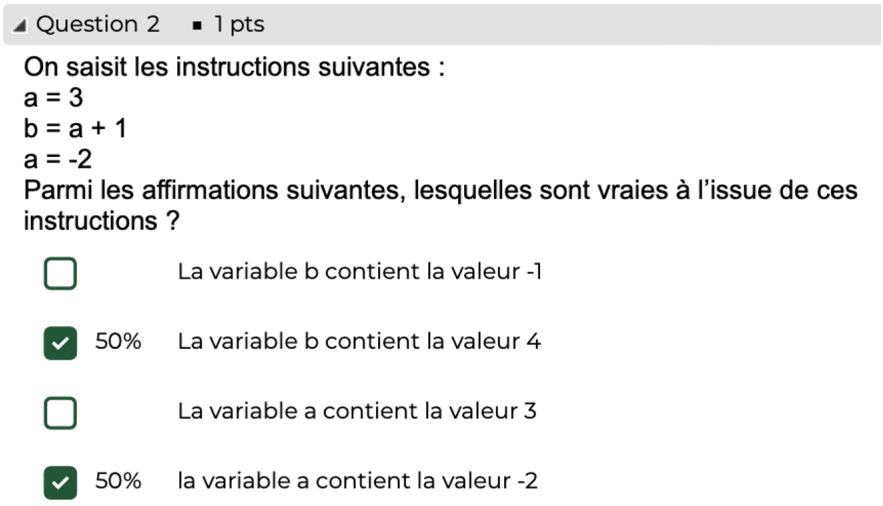
```
1 a = 7
2 b = 3
3 c = a + b
4 c = c + 1
5 b = a - c
```

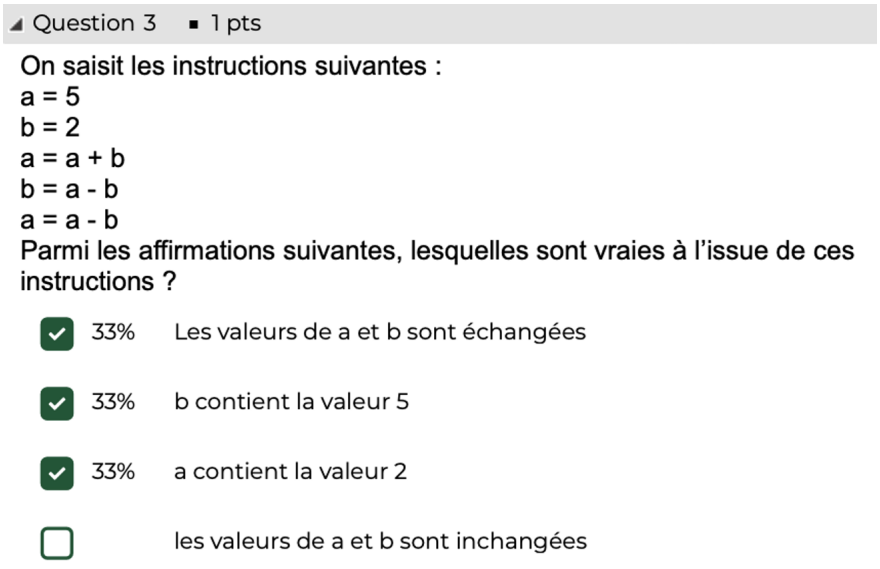
Sur une feuille copiez le tableau suivant et complétez-le :

Contenu variables en...	a	b	c
Ligne 1			
Ligne 2			
Ligne 3			
Ligne 4			
Ligne 5			

Correction du QCM que vous avez fait sur ProNote :







2.4 Les scripts

Un script n'est rien de plus qu'**une succession d'instructions** qui sont exécutées à la suite — dans une console chaque instruction est exécutée lorsque vous passez à

la ligne suivante tandis que dans un script elles le sont toutes sans que vous n'ayez rien à faire pour passer de l'une à la suivante.

Un script peut être **enregistré** pour être **exécuté plusieurs fois** ou dans plusieurs contextes —c'est, au sens où il contient une suite d'instructions mettant en œuvre un algorithme, réellement un **"programme informatique"**.

— Depuis votre ENT, lancez l'application "Capytale".
— Avant de vous lancer n'oubliez pas qu'avec Python on peut convertir (quand c'est possible) des variables d'un type à un autre :

```
# Mettre un nombre au format chaine de caracteres (string)
a = 1984
b = str(a)
c = "Roman de Orwell: " + b
# Ou au contraire:
a = input("Quel est votre age?   ")
b = int(a)
print(b + 10)
# Ou encore en float...
a = input("Que vaut pi avec deux decimales?")
b = float(a)
aire2 = b * 2 ** 2
```

— Rejoignez l'activité que j'ai préparée pour la séance avec le code :
d00f-1846631

2.5 Introduction aux notebooks Jupyter

— Les notebooks Jupyter sont des cahiers électroniques qui, dans le même document, peuvent rassembler du texte, des images, des formules mathématiques et du code informatique exécutable. Ils sont manipulables interactivement dans un navigateur web.
— Ces notebooks sont organisés en cellules qui peuvent être soit du texte (auquel cas on peut y inclure des images ou des formules également) soit du code Python (pour ce cours en tous cas).
— Ces notebooks sont facilement convertibles au format PDF en ligne — ce qui permet de réellement s'en servir comme d'un bloc-notes.
— Tout se passe comme si les cellules de code étaient dans des terminaux les uns à la suite des autres — mais entre elles je peux mettre des instructions et vous pouvez prendre des notes !

Ceci est une cellule de texte

On peut y mettre des titres, des sous-titres...

Des sous-sous-titres etc... Un peu comme en HTML.

C'est d'ailleurs aussi un langage balisé, mais beaucoup plus simple, appelé "markdown" - on y reviendra.

In [7]:
```
# Ici on met une cellule de code - dans laquelle on peut mettre des commentaires
a = 3.1416
# Cette cellule ne renvoie rien - elle ne fait qu'une affectation
```

In [8]:
```
# Celle-ci, en revanche, envoie quelque chose à la sortie
print(a * 2 ** 2)
```

```
12.5664
```

Dans la cellule précédente on a utilisé la formule de l'aire d'un cercle πR^2

In [3]:
```
# Celle-ci afficherait quelque chose si on l'effectuait dans la console
# Dans les notebooks, ceci s'affiche sous forme de "Out"
5+2
```

Out[3]: 7

— Depuis votre ENT, lancez l'application "Capytale".
— Rejoignez l'activité que j'ai préparée pour la séance avec le code :
`ab1d-1883830`

A partir de maintenant nous allons régulièrement utiliser ce format pour les TPs — alors prenez-soin de bien vous y habituer !

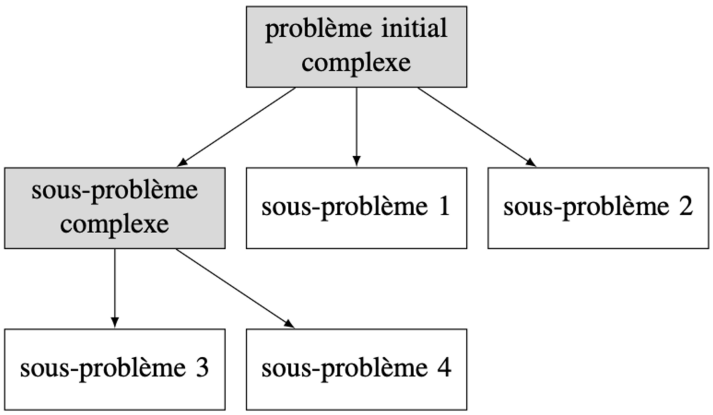
3 Programmation en Python

3.1 Fonctions

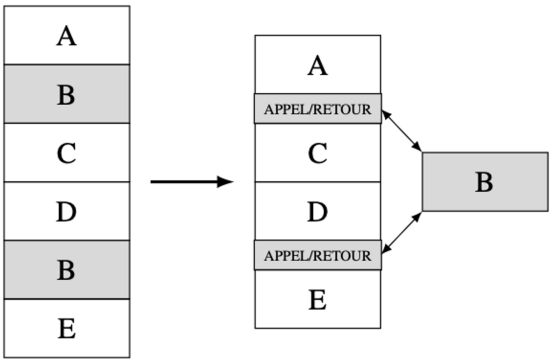
Attention pour commencer : comme pour le "=" et pour la notion de variable, le mot "fonction" n'a pas le même sens en mathématiques et en informatique : dans le premier cas c'est *une relation entre deux ensembles*, dans le second *un procédé de calcul* — comme nous allons le découvrir.

Une **fonction** :
- Est définie par un **nom** ;
- Correspond à une **séquence d'instructions** réalisant une tâche précise ;
- Utilise aucun, un, ou plusieurs "**arguments**". Ces arguments sont les paramètres que la fonction reçoit en entrée ;
- Renvoie généralement **un résultat** ; si elle ne le fait pas, on parle d'une **procédure**.

Les fonctions ont plusieurs utilités :
- Décomposition d'une tâche complexe :



- Réutilisation de code / éviter les répétitions :



- En conséquence : rendre le code plus court et lisible.

Syntaxe en Python :

```
# Definition de la fonction
def nom_fonction(liste_parametres):
        <bloc d'instuctions>
        return resultat

# Appel d'une fonction
resultat = nom_fonction(parametres)

# Appel d'une procedure
nom_procedure(parametres)
```

Attention à l'indentation !!

Considérons le code suivant :

```
1 def conv_minutes(heures, minutes):
2     minutes = minutes + heures * 60
3     return minutes
4
5 resultat = conv_minutes(2, 30)
6
7 print("2 heures 30 font: ", resultat, " minutes")
```

Vérifiez que ce code affichera à l'utilisateur :

```
2 heures 30 font 150 minutes
```

Note : lorsqu'une fonction est appelée, Python réserve pour elle (dans la mémoire de l'ordinateur) **un espace de noms**. Cet espace de noms local à la fonction est différent de l'espace de noms global où se trouvent les variables du programme principal. Dans l'espace de noms local, nous aurons des variables qui ne sont accessibles qu'au sein de la fonction — on parle alors de **variables locales**.

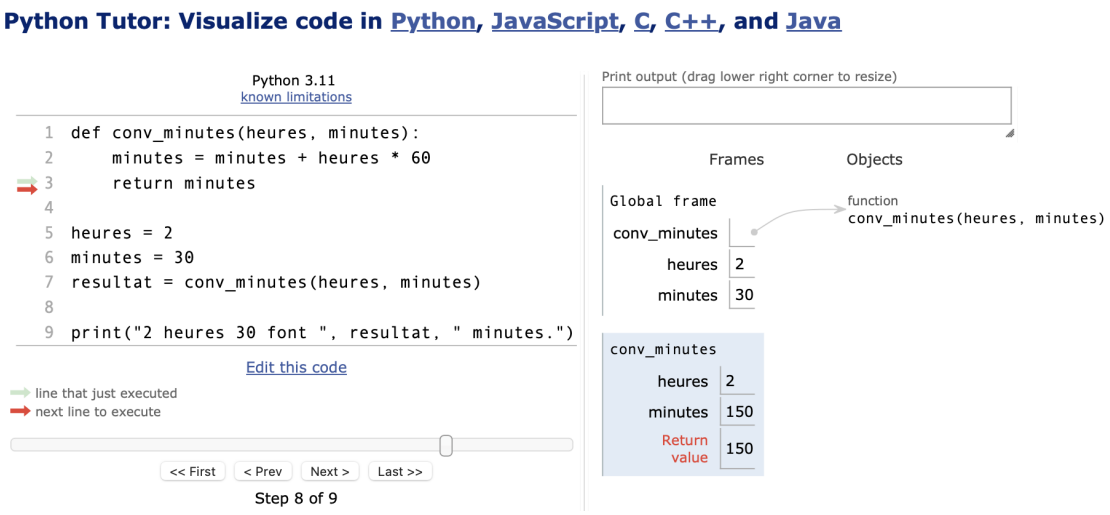
— Rendez-vous sur https://pythontutor.com.
— Cliquer sur "Python".
— Écrire le code suivant :

```
def conv_minutes(heures, minutes):
    minutes = minutes + heures * 60
    return minutes

heures = 2
minutes = 30
resultat = conv_minutes(heures, minutes)

print("2 heures 30 font ", resultat, " minutes.")
```

— Cliquer sur Visualize Execution et exécuter la fonction pas à pas.
— Ouvrez IDLE.
— Écrivez une fonction perimetre_rectangle qui prend en argument deux paramètres — une longueur et une largeur — et qui renvoie le périmètre du rectangle.
— Exécutez-là plusieurs fois et observez les sorties :

```
perimetre_rectangle
perimetre_rectangle()
perimetre_rectangle(3, 5)
perimetre_rectangle(10, 2)
perimetre_rectangle(9)
```

Pour rappel, ce que vous affiche PythonTutor dans ce cas ressemble à ceci :

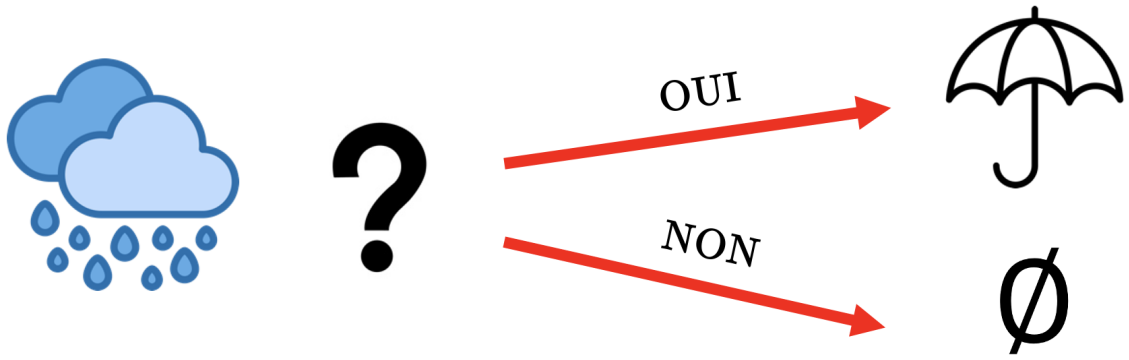


N'hésitez pas à utiliser Python Tutor régulièrement pour visualiser l'organisation de l'utilisation de la mémoire par Python dans les programmes que vous développez — vous verrez, ça vous éclairera souvent sur des bugs que vous ne comprenez pas !

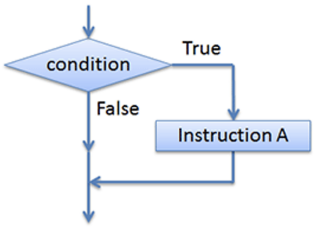
3.2 Conditions & Embranchements

Introduction

Situations classiques de la vie courante — *je dois sortir...*



On a mis en place une **condition** ("s'il pleut") et une **structure d'embranchement** (ce que je fais dans un cas ; ce que je fais sinon).



Remarque : une structure d'embranchement, dans la vie courante comme en programmation, une fois résolue, revient à la poursuite "normale" des actions. Ici :

- S'il pleut — *alors* je prends mon parapluie, *puis* je sors.
- S'il ne pleut pas — *alors* je ne fais rien de spécial, *puis* je sors.

Conditions et tests :

Une **condition** est :

- Un **énoncé** qui peut prendre **deux valeurs et seulement deux valeurs** : Vrai ou Faux.
- Sa **résolution** est donc une **variable booléenne (True / False)**.
- Une **combinaison de tests reliés par des opérateurs logiques**.

Ces tests sont la plupart du temps réalisés à l'aide d'opérateurs de comparaison dont les plus fréquents sont :

Opérateur	Signification
`==`	égal
`!=`	différent de
`<=`	inférieur ou égal
`>=`	supérieur ou égal
`<`	strictement inférieur
`>`	strictement supérieur

Les opérateurs logiques classiques quant à eux sont :

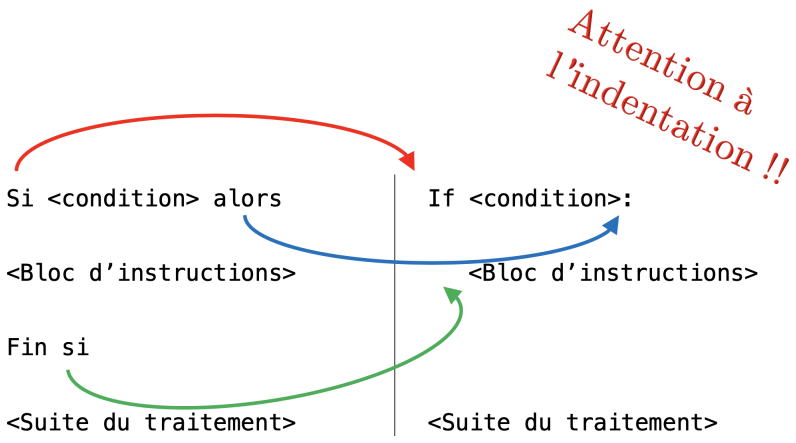
Opérateur	Signification
`not`	"non" logique — négation d'une condition
`or`	"ou" logique — vrai si une de deux conditions au moins est vraie, faux sinon
`and`	"et" logique — vrai si les deux conditions sont vraies, faux sinon

— Lancer IDLE
— Testez dans la console différentes conditions combinant les éléments précédents — par exemple :

```
>> a = 4
>> b = 5
>> a == 4
>> a == 4 or b < 10
>> a == 4 and b < 10
>> a >= 4 or b == 6
>> a < 3 and b == 5
>> not(b == 5)
```

Embranchements :

La syntaxe en python pour réaliser un **embranchement** simple (tel que l'exemple de la pluie plus haut) est la suivante :




```
1  a = -2
2
3  if a <= 0:
4      print("a negatif")
5  print("Fin premier test")
6
7  if a >= 0:
8      print("a positif")
9  print("Fin second test")
```

Puisque les embranchements ne sont parcourus *que* si la condition est remplie, mais que le corps principal du programme (qui dans ce cas en est la partie non-indentée) l'est dans tous les cas, l'affichage en sortie du code ci-dessus sera :

```
a negatif
Fin premier test
Fin second test
```

Python permet également de réaliser des **embranchements avec alternative** :

Attention à l'indentation !!

`Si <condition> alors`	`If <condition>:`
`<Bloc d'instructions1>`	`    <Bloc d'instructions1>`
`Sinon`	`Else:`
`<Bloc d'instructions2>`	`    <Bloc d'instructions2>`
`Fin si`	
`<Suite du traitement>`	`<Suite du traitement>`

```
a = -2

if a > 0:
    print("a positif")
else:
    print("a negatif")
print("Fin du test")
```

Ce qui donne :

```
a negatif
Fin du test
```

Et Python permet enfin d'envisager de multiples alternatives en utilisant la structure dite "sinon si" :

`Si <condition> alors`	`If <condition>:`
`<Bloc d'instructions1>`	`    <Bloc d'instructions1>`
`Sinon si <condition2>`	`Elif <condition2>:`
`<Bloc d'instructions2>`	`    <Bloc d'instructions2>`
`Sinon si <condition3>`	`Elif <condition3>:`
`<Bloc d'instructions3>`	`    <Bloc d'instructions3>`
`(...)`	`(...)`
`Sinon`	`Else:`
`<Bloc d'instructions n>`	`    <Bloc d'instructions n>`
`Fin si`	
`<Suite du traitement>`	`<Suite du traitement>`

— Depuis votre ENT, lancez l'application "Capytale".
— Rejoignez l'activité que j'ai préparée pour la séance avec le code :
`5dc2-1885289`
— Une fois que vous avez fini cette activité je vous propose deux exercices supplémentaires facultatifs :
`d80d-1885381`

ATTENTION : Comme on l'a vu dans l'activité, les calculs algébriques et les tests avec les nombres entiers (type int) sont tous exacts. En revanche, **il n'en est pas de même avec les nombres flottants** (type float) : certains calculs sont exacts et d'autres non, de même pour les tests. Nous verrons plus loin dans le programme pourquoi il en est ainsi (c'est dû à la façon dont les nombres flottants sont représentés en binaire dans l'ordinateur). Ce qu'il faut retenir ici, c'est qu'il ne faut pas faire des tests d'égalité brutaux entre deux nombres flottants.

3.3 Boucles

Boucles itératives

Il nous arrive souvent de devoir répéter plusieurs fois la même opération — dans la vie courante on peut penser à des exercices physiques (des pompes par exemple) ou des actes élémentaires comme mettre un article dans un caddie au supermarché ou encore arroser toutes les plantes d'un jardin. Dans tous ces cas on a une action de base identique qu'on va répéter — soit dans exactement le même contexte (les pompes) soit en le faisant varier légèrement (le supermarché où on ne prend a priori pas n fois le même article).

La même chose se produit en calculs et donc en programmation — par exemple : on place une somme de 1000 euros sur un compte rémunéré à 2% (on multiplie donc chaque année la somme disponible sur le compte par 1,02) et on aimerait savoir la somme dont on disposerait dans 10 ans.

L'approche "brutale" consisterait à procéder ainsi :

```
somme0 = 1000
somme1 = somme0 * 1.02
somme2 = somme1 * 1.02
somme3 = somme2 * 1.02
...
```

On pourrait l'améliorer légèrement en limitant le nombre de variables utilisées :

```
somme = 1000
somme = somme * 1.02
somme = somme * 1.02
somme = somme * 1.02
...
```

Mais dans les deux cas on ferait 11 fois rigoureusement le même calcul — et si soudain on nous demandait combien ça ferait au bout de 100 ans, on devrait le faire 101 fois !

Solution pour éviter cet écueil : les **boucles itératives** ou **boucle pour** qui permet de répéter une séquence d'instructions un nombre fixé de fois :

```
Pour compteur allant de valeur_init à valeur_finale

<Bloc d'instructions>

Fin pour

<Suite du traitement>
```

Ce qui, en syntaxe Python, s'écrit :

```
For compteur in range(valeur_init, valeur finale + 1):

    <Bloc d'instructions>

<Suite du traitement>
```

*(Et, évidemment, **ATTENTION A L'INDENTATION ! !**)*

Quelques remarques au sujet de cette syntaxe :

- Si `valeur_finale < valeur_init` , il n'y a pas d'erreur : la boucle ne s'exécute simplement pas et on passe directement à la suite du traitement.
- En Python, l'instruction `range(valeur_init, valeur_finale + 1)` définit une plage de **valeurs entières consécutives** commençant à valeur_init et se terminant à valeur_finale (cette plage contient donc `(valeur_finale - valeur_init + 1 )` valeurs – ce qui explique le `"+1"` dans la syntaxe. Ainsi :
  - L'instruction `range(1,6)` correspond à la plage de valeurs `{1, 2, 3, 4, 5}`.

— L'instruction `range(0,3)` correspond à la plage de valeurs `{0, 1, 2}`.
— Dans le cas précédent où la valeur initiale est égale à 0, on peut l'omettre en n'écrivant que `range(valeur_finale + 1)` soit, dans ce cas : `range(3)`.

Pour revenir à notre problème des 1000 euros placés sur un compte rémunéré, la fonction suivante va renvoyer la somme demandée :

```
def Calc_Interet():
    somme = 1000
    for i in range(1, 11):
        somme = somme * 1.02
    return somme
```

Attention à la double indentation !!

— Depuis votre ENT, lancez l'application "Capytale".
— Rejoignez l'activité que j'ai préparée pour la séance avec le code :
`1a11-1938165`

ATTENTION : à partir de maintenant les exercices vont relever de la réelle programmation — au sens "résolution d'un problème au moyen d'un programme" ce que vous allez trouver significativement plus difficile que ce que l'on a fait jusqu'à présent. Conseils, si vous avez du mal :
— Prenez le temps de réfléchir mais ne restez pas bloqué·e — n'hésitez pas à demander de l'aide (à moi, à vos camarades...).
— Il n'y a aucun secret : "plus on code, meilleur on sera". Entrainez-vous le plus possible en utilisant les notebooks fournis en séance, mais aussi France IOI (que je regarderai de temps en temps).

Boucles conditionnelles

Reprenons l'exemple de la section précédente du placement de 1000 euros sur un compte rémunéré à 2% (on multiplie donc chaque année la somme disponible sur le compte par 1,02). Cette fois, au lieu de déterminer la somme disponible après un certain nombre d'années, on cherche à calculer le nombre d'années qu'il faudrait attendre avant de disposer d'au moins 2000 euros.

On va donc de nouveau effectuer une boucle sur le calcul `"somme = somme * 1.02"` sauf que cette fois on ne sait pas à l'avance combien de fois on va faire le calcul — ce nombre de fois est même précisément ce que l'on cherche à calculer. Nous allons donc répéter l'opération jusqu'à ce qu'une condition (`"somme >= 2000"`) soit remplie. En algorithmique, cette démarche s'appelle **boucle conditionnelle** ou encore **boucle tant que** :

```
Tant que <condition vraie>

<Bloc d'instructions>

Fin tant que

<Suite du traitement>
```

Ce qui, en syntaxe Python, s'écrit :

```
While <condition vraie>:

    <Bloc d'instructions>

<Suite du traitement>
```

(Et, encore et toujours, ***ATTENTION A L'INDENTATION ! !****)*

Pour revenir à notre problème des 1000 euros placés sur un compte rémunéré, la fonction suivante va renvoyer le nombre d'années :

```
def Calc_Annees():
    annee = 0
    somme = 1000
    while somme < 2000:
        somme = somme * 1.02
        annee = annee + 1
    return annee
```

Attention à la double indentation !!

ATTENTION à la définition des boucles tant que — considérez la fonction ci-dessous :

```
1 def essai_boucle():
2     i = 1
3     while i != 100:
4         i = i + 2
5 print("fonction terminee")
```

Que va-t-il se passer si je fais un appel à cette fonction essai_boucle ?

La condition `i = 100` ne sera ***jamais*** remplie puisque i est initié (ligne 2) à 1 puis est incrémenté dans la boucle (ligne 4) de 2 en 2 — et ne prendra donc jamais que des valeurs impaires. Donc la condition "`i != 100`" sera *toujours* vraie, et on a donc défini **une boucle infinie** — qui soit ne terminera jamais (ce sera le cas dans le cas d'un traitement simple comme celui-ci) et nécessitera une intervention manuelle de l'utilisateur pour être arrêtée, soit va à terme (dans des cas plus complexes) engendrer un dépassement de capacité et donc faire planter le programme. Il faut donc toujours bien vérifier que la condition du "tant que" va, à un moment donné, devenir fausse.

— Depuis votre ENT, lancez l'application "Capytale".
— Rejoignez l'activité que j'ai préparée pour la séance avec le code :
`88a7-1939474`

4 Types de données construits

Jusqu'à présent nous n'avons manipulé que des types de données simples : des entiers, des flottants, des chaines de caractères, des booléens.

En informatique, comme dans la vie courante, ces types de données sont rarement suffisants — il nous est nécessaire de considérer des ensembles de données unis entre eux : c'est ce qu'on appelle des types de données construits (par opposition aux types de données simples, donc) — et c'est ce que nous allons étudier à présent.

4.1 Listes

Introduction

Si on désire étudier l'évolution d'un prix sur plusieurs années, on ne va pas regarder uniquement un prix en particulier, mais l'ensemble des prix sur la période. Il faut donc être capable de stocker toutes ces valeurs dans une même structure afin de pouvoir y accéder sans difficulté.

La structure la plus simple que l'on puisse utiliser est un tableau (en général unidimensionnel, c'est-à-dire avec une seule ligne) :

Prix 1	Prix 2	Prix 3	Prix 4	Prix 5	(...)	Prix n

En Python, un tableau est assimilé à une structure spécifique appelée liste (de type `list`).

Une **liste** est une **collection ordonnée** (ou séquence) d'éléments encadrée par des **crochets**, les éléments étant séparés par des **virgules**.

`[1, 2, -5, 10]` est une liste de quatre éléments ; `["M", "a", "r", "c"]` en est une également ; tout comme `[99, "bonjour", -3, "$"]`.

La **longueur d'une liste**, qu'on peut obtenir avec la fonction `len`, est le nombre d'éléments qui la composent.

`len([1, 2, -5, 10])` renverra la valeur 4.

On peut initier / définir des listes de plusieurs manières :

- **Créer une liste vide :** `Lst1 =[]`
- **Créer une liste "par extension"** :
  - Soit on la définit directement avec les éléments qui la composent :
    `Lst2 =[1, 3, -2, 10]`
  - Soit on y ajoute des éléments au fur et à mesure au moyen de la méthode (ou fonction) `append` qui prend en argument une variable et l'ajoute "au bout" d'une liste :
    ```
    L = [1, 5, 8]
    L.append(150)
    L == [1, 5, 8, 150]
    ```
    renverra `True` parce que le `append` aura ajouté la valeur 150 après le dernier élément de la liste.

— Deux autres approches — **"par transtypage"** et **"par compréhension"** seront couvertes plus loin dans ce cours.

Considérons quelques caractéristiques de ces listes :

— On peut effectuer divers **tests sur des listes** :
  — Test d'**égalité** : `[1, 2, 3] == [1, 2, 3]` renverra `True` puisque les deux listes sont égales.
  — Test d'**inégalité** : `[1, 2, 3] != [1, 3, 2]` et `[1, 2, 3] != [1, 2, 3, 4]` renverront tous les deux `True` puisque les listes ne sont pas strictement égales entre elles.
  — Test d'**appartenance** au moyen du mot clé `"in"` : `2 in [1, 2, 3]` renverra donc `True`.
  — Et à l'inverse, test de "**non appartenance**" en utilisant l'opérateur logique `not` que l'on a déjà vu : `5 not in [1, 2, 3]` renverra `True`.
— Les éléments d'une liste sont des **variables** - on peut donc les manipuler de la même manière que des variables, et notamment les **(ré) affecter**. Par exemple :

```
L = [1, 2, -5, 10]
L[2] = 0
L == [1, 2, 0, 10]
```

renverra `True` puisqu'on aura "remplacé" la valeur -5 par la valeur 0 dans L[2].
— **Indexation** :

Indice	0	1	2	(...)	(N - 1)
Elément	1er	2ème	3ème	(...)	Nème

  — Le premier élément a pour indice 0 ;
  — Le second a pour indice 1 ;
  — (...)
  — Le $n^{ème}$ a pour indice (n-1).
— A l'inverse, pour **trouver un élément** dans la liste :
  — L'élément d'indice j d'une liste L est défini par l'instruction `L[j]`.
  Exemple : on considère la liste `L = [1, 2, -5, 10]`. Alors :
    — `L[0]` correspond à `"1"` ;
    — `L[2]` correspond à `"-5"` ;
    — etc...

  Remarque : si on veut accéder à un indice qui n'existe pas, on déclenche une erreur de type

```
IndexError: list index out of range
```

  c'est-à-dire que l'on a voulu utiliser un indice en dehors de la plage autorisée (de 0 jusqu'à `len` - 1).
  — On peut effectuer des **opérations** sur des listes au moyen d'opérateurs — ***qu'il ne faut pas confondre avec les opérateurs arithmétiques utilisant les mêmes symboles*** :
    — L'opérateur `"+"` est un **opérateur de concaténation** — il permet de rassembler deux listes : l'instruction `[1, 2] + [3]` renvoie la liste `[1, 2, 3]`.

— L'opérateur `"*"` est un **opérateur de duplication** — il permet de dupliquer plusieurs fois une liste pour en créer une autre : l'instruction `[1] * 3` renvoie la liste `[1, 1, 1]` ; l'instruction `[1, 2] * 2` renvoie la liste `[1, 2, 1, 2]`.

— Depuis votre ENT, lancez l'application "Capytale".
— Rejoignez l'activité que j'ai préparée pour la séance avec le code :

`7b7d-2075987`

— *Si vous avez terminé l'activité ci-dessus, penchez-vous sur une série d'exercices supplémentaires plus sophistiqués en utilisant le code :*

28b9-2076098