

INITIATION À LA PROGRAMMATION

Bastien Gorissen & Thomas Stassin

INSTALLER PYTHON

"Apprivoiser le serpent."

POUR FAIRE DU PYTHON, DE QUOI AVONS-NOUS BESOIN ?

- On a besoin de Python (logique)
- Un éditeur de code

Nous allons prendre un peu de temps ensemble pour installer ça et être que tout fonctionne correctement.

INSTALLATION - PYTHON

Il existe plusieurs façons d'installer Python (sur Windows):

- Via le Windows Store
- Via l'installateur dédié

Nous allons utiliser l'installateur dédié, vous pouvez le trouver à:

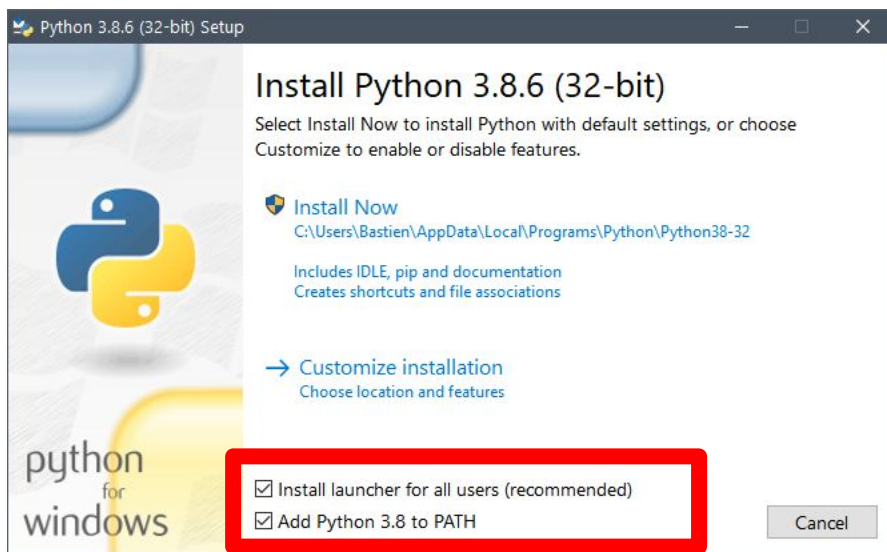
<https://www.python.org/downloads/>

INSTALLATION - PYTHON

Cette formation utilise Python 3.9.1

En installant Python, faites attention à bien cocher la case **Add Python 3.9 to PATH**.

L'installateur va ensuite installer Python, une série de modules (la librairie standard), un éditeur très simple (IDLE), et un installateur de paquets (pip).



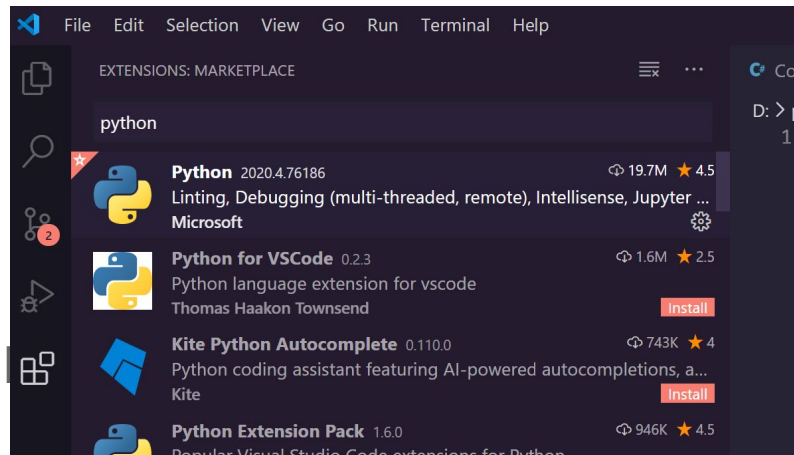
INSTALLATION - VISUAL STUDIO CODE

Vous aurez également besoin d'un éditeur moderne pour éditer votre code. Nous vous conseillons Visual Studio Code, qui contient beaucoup d'outils intéressants, et que vous pouvez télécharger ici:

<https://code.visualstudio.com/>

D'autres options sont possibles (Sublime Text, PyCharm, ...), mais les explications seront données en fonction de VSCode.

Une fois VSCode ouvert, il faut également **installer l'extension Python** pour lui ajouter support du langage.



PREMIER DÉFI

"Guess the number"

PREMIER DÉFI

Le premier programme que nous allons réaliser ensemble est appelé "Guess the number" ou "Devine un nombre".

Comme une image vaut parfois mieux qu'un long discours, voici un gif représentant le programme.

A dark-themed terminal window with a light gray border. The prompt text 'Donnez moi un chiffre entre 1 et 10: |' is displayed in a light gray monospace font at the top left. The rest of the terminal area is empty and dark.

PREMIER DÉFI

Le programme va essayer de faire deviner au joueur un nombre entre 1 et 10.

À chaque tentative, le programme dira au joueur si son nombre est trop petit, trop grand ou bien s'il a deviné la bonne réponse.

Au bout de 3 tentatives, si le joueur n'a pas trouvé, il a perdu.

COMME IL EST ÉCRIT SUR LE GUIDE DU ROUTARD INTERGALACTIQUE...

Pas de panique.

Le but est d'arriver à faire ce programme pas à pas, et de voir un à un les instructions et mécanismes qui vous permettront d'arriver à écrire ce programme.



HELLO WORLD...

"encore lui..."

EST-CE QU'ON PEUT ENVOYER SES PROPRES MESSAGES ?

Une des instructions dont nous avons besoin est un moyen d'afficher du texte à l'écran.

Il existe plusieurs façons de faire "sortir" un message depuis le script Python vers le monde extérieur.

Ce sont des "*instructions de sortie*".

La plus courante est `print()`, qui permet d'écrire dans la console.

Ex. :

```
print("Bravo, vous avez gagné.")
```

ET DONC ON PEUT ÉCRIRE CE QU'ON VEUT ?

Oui, oui ! D'ailleurs...

A vous de jouer !

Utilisez **print()** pour écrire fameux message
"Hello World"

"Prévision : temps..."

... VARIABLE

UNE VARIABLE, C'EST QUOI ?

Il est souvent utile de garder une donnée en mémoire pour la réutiliser.

Ex.:

- Les points de vie du héros
- Le nom d'un utilisateur
- Un nombre à faire deviner
- La réponse qu'un utilisateur a donnée
- ...

Pour ce faire, on utilise des "*variables*".

C'EST JUSTE UNE VALEUR ?

Voici une définition de ce qu'est une variable:

Une variable est zone de la mémoire libellée et dédiée à contenir des informations.

Dans le code suivant:

```
number_of_cups = 6
```

On mets la valeur **6** dans la variable **number_of_cups**.

C'EST JUSTE UNE VALEUR ?

```
number_of_cups = 6
```

Dès le moment où cette ligne de code est exécutée par Python, une zone de la mémoire est dédiée à accueillir la valeur **6**.

Python l'associera au nom **number_of_cups**.

Cette opération est nommée l'affectation.

L'affectation est donc l'opération qui lie une valeur à une variable.

AFFICHER UNE VALEUR:

Une des manières d'afficher le contenu d'une variable à l'écran est d'utiliser la fonction `print` vue précédemment.

```
age = 38  
print(age)
```

À l'écran va s'afficher `38`.

STOCKER LE RÉSULTAT D'UN CALCUL ?

Lors de l'affectation vous pouvez aussi affecter une expression (un "calcul").

```
number = 6 + 3
```

Python fera le calcul pour vous et stockera 9.

Dans le même ordre d'idée, vous pouvez aussi utiliser une autre variable lors de l'affectation. Python fera aussi le calcul, substituant la valeur à la variable.

```
number = 30
```

```
other_number = 60 - number
```

AFFECTATION EN SÉRIE.

`a = 6`

`b = 8`

`c = a + b`

`a = c + 3`

`print(c)`

`print(a)`

Quel seront les chiffres affichés ?

AFFECTATIONS EN SÉRIE.

a = 6

b = 8

c = a + b

a = c + 3

print(c)

print(a)

a	b	c
6		
6	8	
6	8	14
17	8	14
17	8	<u>14</u>
<u>17</u>	8	14

SELF-AFFECTATION

Comme nous l'avons vu, il est possible d'utiliser une variable lors de l'affectation. Et donc rien ne nous empêche donc de faire ça.

a = 2

a = a + 1

Dans ce cas, comme précédemment, la valeur sera substituée à la variable et comme **a** vaut **2**, la valeur qui sera affectée à **a** sera **2 + 1** donc **3**.

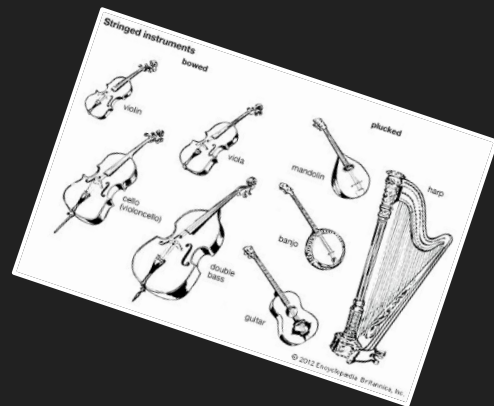
SELF-AFFECTATION

Affecter une variable en utilisant sa valeur est appelé ***incrémentation*** ou ***décrémentation*** en fonction de si la variable augmente sa valeur ou la baisse.

Nous reverrons ce genre de code très souvent dans le futur.

ON VA PARLER DE STRING

"Bel enchaînement."



EST-CE QU'ON PEUT METTRE DU TEXTE DANS UNE VARIABLE ?

On peut, et c'est un cas assez courant.

En informatique, on appelle un bout de texte une "*chaîne de caractères*" ou "*character string*" (ou "*string*") en anglais.

Ex.:

```
message = "Ceci est mon message."
```

La variable `message` contient donc le texte `"Ceci est mon message."` Plutôt logique, non ?

QUEL AVANTAGE ?

On peut utiliser des "*string*" pour manipuler du texte dynamiquement.

Par exemple, prenons le petit script suivant :

```
hero_name = "Brutor"  
message = hero_name + ", il reste des monstres à vaincre."  
print(message)
```

Et donc le message affiché sera **"Brutor, il reste des monstres à vaincre."**

On appelle ceci de la ***concaténation*** de chaîne.

PEUT-ON CONVERTIR QUELQUE CHOSE EN STRING ?

Il existe une fonction en Python qui permet la "*conversion*" de données en texte.

C'est `str()`.

Ex.:

```
a = 3
```

```
b = str(a)
```

a contient le nombre 3, et **b** contient le texte "3".

ON COMBINE.

```
a = "Number "
```

```
b = "Lucky "
```

```
c = b + a
```

```
a = 7
```

```
c = c + str(a)
```

```
print(c)
```

Quel seront le message affiché?

ON COMBINE.

```
a = "Number "
```

```
b = "Lucky "
```

```
c = b + a
```

```
a = 7
```

```
c = c + str(a)
```

```
print(c)
```

a	b	c
"Number "		
"Number "	"Lucky "	
"Number "	"Lucky "	"Lucky Number "
7	"Lucky "	"Lucky Number "
7	"Lucky "	"Lucky Number 7"
7	"Lucky "	<u>"Lucky Number 7"</u>

ENTRÉE...

"Là où l'on interagit avec l'utilisateur"

ENTRÉE PRINCIPALE

Parfois, vous ne voudrez pas seulement afficher, mais aussi récupérer des données, comme par exemple, les propositions de nombres pour la devinette.

Pour ce faire, on utilise une instruction dite d'entrée.

La plus courante se nomme **input()**.

Elle s'utilise comme suit:

```
name = input("Quel est ton nom? ")
```

ENTRÉE PRINCIPALE

```
name = input("Quel est ton nom? ")
```

`input` va réaliser plusieurs choses:

- Afficher la chaîne de caractères "Quel est ton nom? "
- Attendre que l'utilisateur écrive quelque chose au clavier, et confirme en appuyant sur Enter.
- Renvoyer ce que l'utilisateur a écrit.

Dans la ligne ci-dessus, la valeur retournée est ensuite mise dans la variable **name**.

PETIT EXERCICE

Créer un programme qui demande à l'utilisateur son prénom et qui lui dit lui "Bonjour" en l'appelant par son prénom.

```
Quel est ton prénom: |
```

ENTRÉE VIP

Attention: **input()** renvoie toujours une valeur de type **string**.

Si vous avez besoin d'autre chose, vous devrez procéder à une conversion.

On appelle cette conversion un **casting**.

CONVERSION

"Le cas Sting"

CASTING

Le ***casting*** est le processus de conversion d'un type de données vers un autre.

Il s'agit d'une généralisation de ce que vous avez vu avec la fonction **`str()`**.

Par exemple, pour convertir une chaîne de caractères en nombre entier :

```
a = "11"
```

```
a = int(a)
```

CASTING

Il suffit donc d'utiliser la fonction portant le nom du type vers lequel vous voulez convertir vos données.

Les types de bases de Python sont les suivants:

- **int** pour les nombres entiers.
- **float** pour les nombres à virgule.
- **str** pour les chaînes de caractères.
- **bool** pour les variables booléennes (voir plus loin).

CASTING

Évidemment, le casting ne fonctionnera que si la conversion a du sens.

`int("waaaaah")` : Python n'arrivera pas à transformer "waaaaah" en nombre entier, et il va entrer en erreur. Il vous affichera quelque chose comme ceci:

```
>>> int("waaaaah")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'waaaaah'
```

Ce qui (une fois traduit) est assez explicite comme message d'erreur.

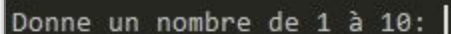
GUESS THE NUMBER

"First step"

PREMIER PAS

Vous avez ici tous les outils pour demander à votre utilisateur de choisir un nombre entre 1 et 10 et de récupérer le nombre dans une variable (en le convertissant).

Note: Vous pouvez afficher le nombre entré par l'utilisateur pour vérifier que tout s'est bien passé, mais cette partie sera retirée du programme final.

A screenshot of a terminal window with a dark background. The text "Donne un nombre de 1 à 10: |" is displayed in a light gray monospace font. The vertical bar indicates a cursor position at the end of the line.

```
Donne un nombre de 1 à 10: |
```




"Comparaison n'est pas raison."

COMMENT FAIRE INTERAGIR 2 VARIABLES ?

Nous l'avons vu via les flowcharts: en programmation, il est possible de comparer des valeurs. Dans les premiers exercices, nous avons élaboré un programme qui comparait l'âge contenu dans une variable avec la valeur 18, pour vérifier le comportement à adopter.

La question posée était: "Est-ce que la variable **age** a une valeur plus petite que **18** ?"

En d'autres termes : **age** < **18** ?

Ceci est une ***comparaison***.

COMMENT RÉCUPÉRER LE RÉSULTAT D'UNE COMPARAISON ?

Le résultat d'une comparaison peut être stocké dans une variable.

Ex. :

```
a = 3  
comp = a < 10  
print(comp)
```

Dans ce cas la valeur affichée sera **True**

Il y a deux valeurs possible pour une comparaison:

True ou **False**

TRUE ET FALSE ? KÉZAKO ?

Souvent, en Python, la réponse se trouve dans la traduction des mots utilisés.

True ↔ Vrai

False ↔ Faux

Python a un type (comme **int**, **str**, ...) qui correspond à une valeur vraie ou fausse, c'est le type **bool**, ou booléen.

Nous allons voir comment nous en servir.

ON PEUT FAIRE QUOI COMME TYPE DE COMPARAISON ?

Excellente question ! Il existe un nombre varié de comparaisons.

- $<$: "plus petit que"
- $<=$: "plus petit ou égal à"
- $>$: "plus grand que"
- $>=$: "plus grand ou égal à"
- $==$: "égal à"
- $!=$: "différent de"

La plupart de ces opérateurs devrait vous rappeler des souvenirs de cours de maths.

TRUE OR FALSE

A votre avis, les expressions suivantes sont-elles **True** ou **False** ?

18 < 12	?
55 > 30	?
5 >= 5	?
7 > 7	?
20 != 10 + 5 + 4 + 1	?
30 == 15 * 0 + 15 * 2	?

TRUE OR FALSE

Réponses:

18 < 12	False
55 > 30	True
5 >= 5	True
7 > 7	False
20 != 10 + 5 + 4 + 1	False
30 == 15 * 0 + 15 * 2	True

EST-CE QU'IL Y A DES PIÈGES ?

Il faut faire attention à certains cas particuliers:

`4 < 4` par rapport à `4 <= 4`

`"message" < "texte"`

`3 > "texte"`

`==` qui est différent de `=` (affectation)

Mais dans l'ensemble ça fait ce qui marqué sur l'étiquette...

OR AND NOT ?

"Swimming bool."

QUE VA-T-ON FAIRE AVEC DES TRUE/FALSE ?

Comme nous l'avons vu, Python a donc un type qui correspond à Vrai/Faux.

C'est une notion qui vient du monde de la logique.

Comme pour les nombres ou le texte, il existe toute une série d'opérations qu'on peut effectuer spécifiquement sur des variables ou des valeurs de type **bool**.

Ca porte le nom d'**Algèbre Booléenne**, nommée d'après celui qui l'a introduite dans le discours mathématique: Georges Boole.

LE NOM N'INSPIRE PAS CONFIANCE...

Il s'agit surtout de "codifier" une série de notions que vous rencontrez finalement tout le temps dans la vraie vie.

Par exemple :

"Je voudrais un t-shirt rouge **OU** avec un logo Star Wars."

"J'ai envie de partir loin **ET** au soleil."

"Je n'ai **PAS** de poisson rouge."

ÇA SEMBLE DE FAIT LOGIQUE.

Comme tout bon mathématicien, George Boole a trouvé tout un tas de règles et de théorèmes spécifiques à ce qui est une véritable branche des mathématiques.

Mais...

Nous allons nous concentrer sur trois opérateurs simples qu'on rencontre tout le temps en programmation :

Le **NOT**, le **AND** et le **OR**.

AND ?

Le **and**, c'est la conjonction. Ou plus sobrement, le "et". Son résultat est **True** si et seulement si 2 conditions sont vraies.

True and True	True
True and False	False
False and True	False
False and False	False

Vous pouvez aussi voir le résultat du **and** comme ceci: dès lors qu'une des valeurs est fausse, l'ensemble est faux.

OR ?

Le **or**, c'est la disjonction. Ou encore, le "ou". Le résultat est **True** si au moins une des 2 conditions est vraie.

True or True	True
True or False	True
False or True	True
False or False	False

Vous pouvez aussi voir le résultat du **or** comme ceci: l'ensemble est faux si et seulement si toutes les conditions sont fausses.

NOT ?

Le **not**, c'est la négation. Comme dans une phrase, quand vous faites une négation, vous inversez les valeurs de vrai ou faux.

not True	False
not False	True

EXERCICES :D

Quelle sera la valeur des expressions suivantes ?

`a = 3`

`b = 5`

`(a == b) or (a < b)`

`(b >= a) and (a != b)`

`(not (a > 0)) and (b == 5)`

`not(a == 3 or b == 5)`

`not(a == 3) and not(b == 5)`

?
?
?
?
?

EXERCICES :D

Réponses:

`a = 3`

`b = 5`

`(a == b) or (a < b)`

`(b >= a) and (a != b)`

`(not (a > 0)) and (b == 5)`

`not(a == 3 or b == 5)`

`not(a == 3) and not(b == 5)`

True
True
False
False
False

ET SI ...

"If only..."

QUAND EST-CE QU'ON FAIT UN TRUC UTILE AVEC TOUT ÇA ?

Maintenant !

Nous avons besoin de vérifier si le nombre donné par l'utilisateur est plus grand, plus petit ou égal à la bonne réponse.

Pour ceci, on utilise une instruction dite de "*contrôle de flux*".

Ca revient à dire : "**Si** une certaine condition est remplie, **alors** fait quelque chose."

ET EN PYTHON ?

Le "si", en Python, ça se dit "**if**" (c'est étonnamment proche de l'anglais, le Python, au final...)

```
a = 3
if a < 10:
    print(str(a) + " est plus petit que 10 !")
    a = 10
```

C'est aussi simple que ça !

Attention à deux choses : les ":" et l'indentation du code à l'intérieur du **if** !

ET AVEC LES BOOLÉTRUCS ?

Un `if`, schématiquement, c'est :

if condition:

print("condition vérifiée")

la ***condition*** peut être n'importe quelle valeur de type **bool**, ce qui veut dire :

`a > 3 and b <= 10` → OK !

`not (name == "Roger")` → OK !

On peut mettre exactement la condition qu'on veut !

EXERCICES :D

Écrivez un programme qui demande à l'utilisateur d'entrer un mot de passe.

Si le mot de passe n'est pas "**Pyth0n**", le programme affichera: "**Code erroné.**"

```
Entrez le mot de passe:
```

AVEC DES SI...

"if, version full option."

UN "SI", ÇA DEMANDE SOUVENT UN "SINON", NON ?

Un `if` en Python peut contenir une clause "sinon" (en Anglais, `else`)

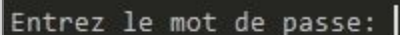
```
if condition:
    print("condition remplie")
else:
    print("condition non-remplie")
```

Il est important de se souvenir que c'est soit le `if` (si la condition est remplie) soit le `else`, mais pas les deux.

ENCORE LE PASSWORD

Reprenez votre programme qui demande à l'utilisateur d'entrer un mot de passe.

Si le mot de passe est "Pyth0n", le programme affichera: "**Code valide.**",
sinon il affichera "**Code erroné.**".

A dark rectangular box representing a terminal window. Inside, the text "Entrez le mot de passe: " is followed by a vertical cursor bar.

```
Entrez le mot de passe: |
```

ET SI ON A PLUSIEURS CONDITIONS À TESTER ?

Une dernière extension du `if` existe si vous avez plusieurs conditions à tester :

```
a = 3
b = 5
if a < b:
    print(str(a) + " est plus petit que " + str(b))
elif a == b:
    print(str(a) + " est égal à " + str(b))
else:
    print(str(a) + " est plus grand que " + str(b))
```

ET SI PLUSIEURS CONDITIONS SONT REMPLIES ?

```
a = 3
b = 5
if a < b:
    print(str(a) + " plus petit que " + str(b))
elif a < 10:
    print(str(a) + " est plus petit que 10")
else:
    print(str(a) + " est plus grand que " + str(b))
```

Attention, seul le bloc de code correspondant à la première condition remplie sera exécuté!

LE HASARD FAIT
BIEN LES CHOSES

"[insert random joke here]"

NOMDRA

Beaucoup de jeux (par exemple) mettent à profit la notion de hasard pour offrir au joueur une expérience toujours renouvelée.

Bien entendu, derrière, ne se cache rien d'autre qu'une instruction spécifique destinée à générer des valeurs aléatoires.

En Python, ces fonctions sont reprises dans la librairie **random**, et nous allons surtout nous intéresser à la fonction **randint()**, qui sert à générer un entier aléatoire.

MARDON

Pour pouvoir utiliser `randint()`, nous allons devoir demander à Python de l'importer.

Pour ce faire, il faut placer en début de script une ligne telle que:

```
from random import randint
```

Ensuite, vous pouvez utiliser `randint()` comme toute autre fonction de Python:

```
d6 = randint(1, 6)
```

La notion d'import sera détaillée plus tard, mais nous en avons besoin pour pouvoir continuer.

RONDMA

```
d6 = randint(1, 6)
```

La fonction **randint()** doit recevoir 2 arguments:

- Un minimum (ici, 1)
- Un maximum (ici, 6)

...et génère un nombre entier compris entre le minimum et le maximum (compris).

Il existe bien d'autres fonctions dans la librairie **random**, mais nous aurons l'occasion de les voir plus tard !

GUESS THE NUMBER

"Second step"

SECOND PAS

Vous avez maintenant les connaissances nécessaires pour:

- Faire en sorte que le programme génère aléatoirement un nombre à deviner entre 1 et 10.
- Demander à l'utilisateur de donner un nombre entre 1 et 10.
- Indiquer à l'utilisateur si son nombre est inférieur, supérieur ou égal au nombre à deviner

SECOND PAS

```
Donnez moi un chiffre entre 1 et 10: |
```

L'ABSENCE DE VALEUR

...est une valeur

NULL

Les informaticiens ont très vite éprouvé le besoin de représenter l'absence de valeur.

Il est important de différencier, par exemple, le fait qu'une variable soit égale à 0 (qui est une valeur comme une autre) du fait qu'une variable soit vide.

Dans le cas où une variable n'a pas de valeur, en informatique, on dira qu'elle est ***NULL***.

NONE

Quand une variable est **null**, cela veut dire qu'elle n'a pas de valeur.

Pour représenter une valeur null en Python nous utiliserons l'expression **None** (attention à la majuscule).

```
answer = None
```

Dans cette ligne de code la variable **answer** sera donc **null**.

C'est par exemple utile quand vous voulez initialiser une variable, mais ne savez pas quelle valeur elle doit avoir au départ.

BOUCLES

"While E. Coyote."

L'ORDINATEUR, CE TRAVAILLEUR INFATIGABLE...

Les ordinateurs sont particulièrement doués pour faire des tâches répétitives (rapidement).

Nous allons voir une façon économique de demander à l'ordinateur de répéter un morceau de code un nombre de fois donné.

On appelle ça une **boucle**.

LA BOUCLE LOGIQUE OU BOUCLE WHILE

Il y a 2 types de boucles:

- Les boucles *logiques*
- Les boucles *arithmétiques*

Nous allons commencer par le premier type.

On utilise le nom "logique", car c'est une boucle qui va s'exécuter "*tant que*" (**while**) une certaine condition est vérifiée.

TANT QUE...

Comme son nom l'indique (en anglais du moins), la boucle **while** bouclera tant que... mais "tant que" quoi?

Et bien tant que la condition qui la suit est *vraie*.

```
a = 0
while a < 3:
    print("a est plus petit que 3")
    a = a + 1
```

Dans cet exemple, la boucle bouclera tant que a est strictement plus petit que 3.

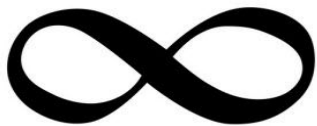
Evidemment, comme pour un **if**, il y a un bloc de code indenté, et contrôlé par cette boucle.

VERS L'INFINI ET AU DELÀ...

```
number = 20  
while number > 10:  
    print("Répétition...")
```

Que se passerait-il si le code ci-dessus était exécuté ?

(Si jamais vous n'avez pas la réponse, vous pouvez tester et constater.)



La condition **number** > **10** sera toujours vraie et donc la boucle continuera, encore...

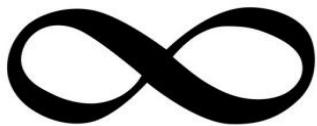
et encore...

et encore...

...

...

et encore...



et encore...

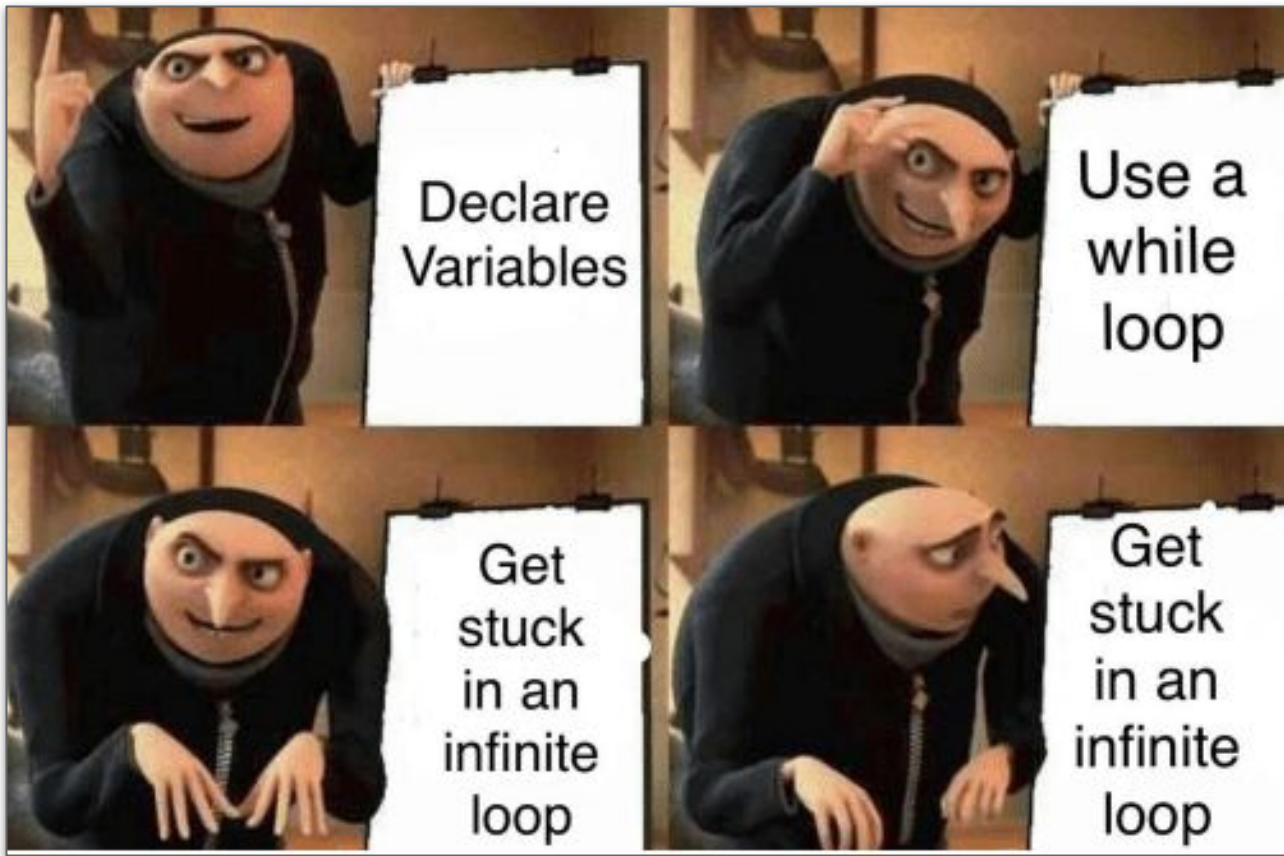
et encore...

...

...

Jusqu'à l'infini...

Le cas de la boucle infinie est bien connu des programmeurs et a déjà dû causer des crises de nerfs.



ET SI JAMAIS ÇA ARRIVE?

Il y a une échappatoire.

En Python, on peut interrompre l'exécution d'un code en pressant les touches **Ctrl + C**

Dans ce cas, une erreur ***KeyInterrupt*** apparaîtra, mais au moins le code s'arrête :)

D'AILLEURS SOYONS FOUS

Si ce n'est pas déjà fait, testez le code suivant:

```
while 1 == 1:  
    print("Vous aimez le comique de répétition ?")
```

Et testez **Ctrl+C**.

Donc, maintenant que nous savons tout ça?

GUESS THE NUMBER

"Last step"

DERNIER PAS

Vous avez maintenant les connaissances nécessaires pour:

1. Faire en sorte que le programme génère aléatoirement un nombre à deviner entre 1 et 10.
2. Demander à l'utilisateur de donner un nombre entre 1 et 10.
3. Indiquer à l'utilisateur si son nombre est inférieur, supérieur ou égal au nombre à deviner.
4. Si la réponse est bonne le programme s'arrête et félicite le joueurs.
5. Si la réponse est mauvaise le programme reprendra au point 2, sauf si c'est la troisième mauvaise réponse, dans ce cas il donnera la bonne réponse à l'utilisateur.

DERNIER PAS

Il est évident que vous aurez besoin d'une boucle while pour ça (mais pas seulement), réfléchissez donc bien à sa condition.

Pour rappel: voici le résultat attendu



```
Donnez moi un chiffre entre 1 et 10: |
```

MASTERMIND

"Eminence grise"

UN NOUVEAU DÉFI

Après notre premier jeu de devinettes, nous allons en réaliser un second, connu sous le nom de "Mastermind".

Dans le jeu de société, un des joueurs choisit un code composé de 4 pions de couleur, et son adversaire doit essayer de déduire le code à l'aide d'informations: à chaque essai, le décodeur sait combien de ses pions sont bien placés, et combien sont de la bonne couleur, mais à la mauvaise place.



UN NOUVEAU DÉFI

Notre version va fonctionner comme le jeu physique, avec toutefois quelques adaptations:

- L'ordinateur (aka, Python) va choisir le code, et l'utilisateur va essayer de le deviner.
- Au lieu d'utiliser des pions de couleur, nous allons utiliser des lettres. Comme le jeu a normalement 6 couleurs, nous allons utiliser les lettres de "a" à "f".

DÉCOUPAGE

Comme pour le "Guess the Number", nous allons découper le travail en plusieurs sous-parties et les développer progressivement.

1. Générer le code à deviner
2. Demander une tentative de décodage au joueur
3. Vérifier quelles lettres proposées par le joueur sont correctes
4. Mettre le tout dans une boucle
5. Gérer les lettres mal placées
6. Gérer le nombre de tentatives et afficher un message de victoire/défaite

PREVIEW

Voici un exemple du résultat que nous obtiendrons à la fin:



LETTRES POSSIBLES

Pour commencer, nous allons générer un code de 4 lettres choisies au hasard.

Pour y arriver, nous allons avoir besoin de créer des "listes".

Nous allons devoir dire à Python parmi quelles lettres il doit choisir, mais aussi stocker la liste des lettres choisies.

Let's do this!

LISTES

"To-do: listes"

LISTES

Jusqu'ici, vous avez utilisé des variables qui contiennent une valeur.

Mais, dans beaucoup de cas, nous aurons besoin de stocker plusieurs valeurs liées ensemble.

Ou d'être capables de stocker un ensemble de données qui peut s'étendre ou diminuer au cours de l'exécution de votre programme.

Ex : stocker une liste d'achats dans un caddy, une liste des utilisateurs connectés, ou encore un ensemble de réponses à un questionnaire...

LISTES: CRÉATION

Pour créer une liste, il suffit d'utiliser des crochets : []

Entre les [], vous pouvez écrire les différents éléments de la liste, séparés par des virgules.

```
word_list = ["I", "love", "Python"]
```

```
number_list = [4, 4, 4, 7, 1, 9]
```

```
empty_list = []
```

LISTES: CRÉATION

```
word_list = ["I", "love", "Python"]
```

```
number_list = [4, 4, 4, 7, 1, 9]
```

```
empty_list = []
```

Comme vous le voyez:

- Vous pouvez mettre ce que vous voulez comme type de donnée dans une liste
- Chaque variable contient une liste, qui elle-même contient plusieurs valeurs
- `[]` dénote une liste vide, qui ne contient aucun élément (mais est quand même une liste)

MIX IT UP

Vous pouvez également mélanger les types dans une même liste.

```
value = 42
text = "Some text"
numbers = [1, 2, 3]
complete_list = [None, value, text, numbers]
```

Ce code générera la liste suivante, contenant 4 éléments:

```
[ None, 42, "Some text", [1, 2, 3] ]
```

MIX IT UP

Vous pouvez également mélanger les types dans une même liste.

```
value = 42
```

```
text = "Some text"
```

```
numbers = [1, 2, 3]
```

```
complete_list = [None, value, text, numbers]
```

Ce code générera la liste suivante, contenant 4 éléments:

```
[ None, 42, "Some text", [1, 2, 3] ]
```

MASTERMIND

"First step"

LISTE DE LETTRES

Vous avez maintenant plus qu'assez d'informations pour pouvoir créer la liste des lettres pour notre jeu de Mastermind.

- Créez une liste qui contient les lettres de "a" à "f" (6 éléments au total)
- Affichez la liste avec un `print()` (temporairement, question de voir que tout fonctionne correctement)

```
['a', 'b', 'c', 'd', 'e', 'f']
```


INDEX

"Chercher les indices"

MISE À L'INDEX

Maintenant que vous savez créer des listes, l'étape suivante consiste à accéder à leurs éléments.

La façon la plus courante pour aller récupérer l'un des éléments d'une liste est l'utilisation d'un **index**.

C'est un nombre qui désigne la position d'un élément dans la liste.

```
words = ["I", "really", "love", "Python"]  
print(words[2])
```

...va imprimer "love" à la console.

EUH...

..."love" n'était pas à l'index 3 ?

Non ! En Python, on commence à compter à partir de zéro !

`words[0]` → Premier élément

`words[1]` → Deuxième élément, ...

index	0	1	2	3
<code>words = [</code>	<code>"I",</code>	<code>"really",</code>	<code>"love",</code>	<code>"Python",</code>

]

INDEX MINIMUM ET MAXIMUM

Les indexes sont toujours entre **0** et "**le nombre d'éléments dans la liste - 1**".

Pour connaître le nombre d'éléments dans une liste, vous pouvez utiliser la fonction `len()`.

```
users = ["Jo", "Alex", "Graham", "Kathleen"]  
count = len(users)  
print(count)
```

L'index maximum de la liste sera donc `len(users) - 1` !

EXERCICE

- Créez une liste contenant 6 éléments.
- Utilisez `randint()` pour choisir un nombre aléatoire.
- Affichez la liste complète.
- Affichez à la console l'élément de la liste se trouvant à l'index correspondant au nombre aléatoire choisi au point précédent.

Attention : veuillez à bien choisir les bornes pour le `randint()` !

```
['Ian', 'Cori', 'Alex', 'Kathleen', 'Graham', 'Heather']  
Kathleen
```

COMPTER À L'ENVERS

Dans certains cas, il peut être pratique de commencer à énumérer les éléments à partir de la fin de la liste. Vous pouvez faire ça en utilisant des index négatifs.

`words[-1]` → Dernier élément

`words[-2]` → Avant-dernier élément...

index	-4	-3	-2	-1
<code>words = [</code>	<code>"I",</code>	<code>"really",</code>	<code>"love",</code>	<code>"Python",]</code>

SLICING AND DICING

Il est également possible d'accéder à plusieurs éléments d'une liste en même temps (et ainsi créer une nouvelle liste).

Il vous suffit de donner un index de départ, et un index de fin, et Python va extraire une sous-partie de la liste.

```
words = ["I", "really", "love", "Python"]  
print(words[1:3])
```

Ceci va renvoyer une sous-liste : **["really", "love"]**

On reçoit donc les éléments à partir de l'index 1 (compris) jusqu'à l'index 3 (non-compris).

SLICING AND DICING

Si vous omettez l'un des deux index, Python ira jusqu'à la fin de la liste dans la direction omise:

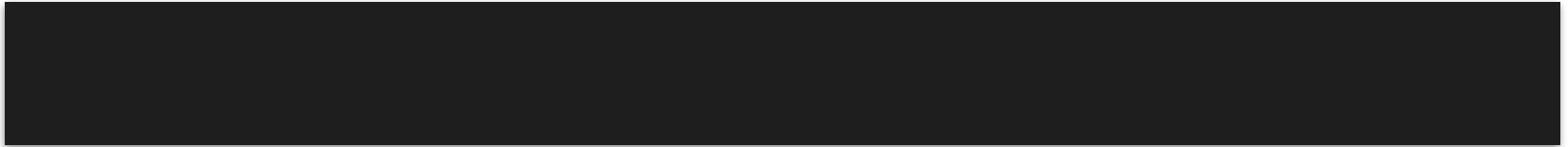
```
words[:3]
```

```
words[1:]
```

Rappel : Quand on spécifie une limite, Python va chercher les éléments depuis le premier index (compris), jusqu'au dernier index (non-compris).

EXERCICE

- Créez une liste contenant 6 éléments.
- Demandez à l'utilisateur un nombre entre 0 et la longueur de la liste - 1.
- Affichez la liste complète.
- A l'aide de deux autres `print()`, affichez d'abord les éléments entre le début de la liste et le nombre choisi (non compris), et ensuite les éléments de la liste entre le nombre choisi (compris) et la fin de la liste.
- Approfondissement: assurez-vous que l'utilisateur donne un nombre valide.



AJOUT / SUPPRESSION

"Modifier le contenu"

MODIFIER UNE LISTE

Nous avons créé des listes, et accédé à leurs éléments, mais souvent, vous aurez besoin de modifier le contenu d'une liste.

Par exemple :

- Ajouter un nouvel élément (un nouvel utilisateur se connecte)
- Supprimer un élément (vous avez traité un élément)
- Remplacer un élément (l'utilisateur veut modifier une réponse qu'il a donnée)
- ...

REEMPLACER UN ÉLÉMENT

Pour remplacer un élément, il suffit de lui assigner une nouvelle variable, en utilisant les [] et un index:

```
numbers = [1, 2, 3, 4]  
numbers[2] = 100  
print(numbers)
```

Affichera:

```
[1, 2, 100, 4]
```

AJOUTER UN ÉLÉMENT: APPEND()

Pour ajouter un élément à une liste, le cas le plus fréquent consiste à l'ajouter à la fin de la liste existante avec `append()` (notez le `words.` avant `append()`) :

```
words = ["I", "really", "love", "Python"]  
words.append("so")  
words.append("much")  
print(words)
```

Affichera donc:

```
["I", "really", "love", "Python", "so", "much"]
```

EXERCICE

- Créez une liste vide.
- Demandez à l'utilisateur d'entrer un mot au clavier.
- Tant que le mot entré n'est pas "stop", ajoutez-le à la liste, et redemandez un nouveau mot à l'utilisateur.
- Affichez tous les mots entrés par l'utilisateur.

```
PS C:\Users\kadom\Dropbox\My PC (DESKTOP-B4KCKJH)\Documents\Interface3\Python> 
```

AJOUTER UN ÉLÉMENT: INSERT()

Une autre façon d'ajouter un élément à une liste est de dire à Python à quel index vous voulez insérer le nouvel élément.

```
words = ["I", "really", "love", "Python"]  
words.insert(1, "actually")  
print(words)
```

Affichera :

```
["I", "actually", "really", "love", "Python"]
```

Le nouvel élément se trouve bien à l'index 1, le reste ayant été décalé.

EXERCICE

- Créez une liste vide.
- Demandez à l'utilisateur d'entrer un mot au clavier.
- Tant que le mot entré n'est pas "stop":
 - Demandez à quel index le mot doit être placé dans la liste
 - Ajoutez le mot à la liste, à l'index demandé
 - Redemandez un nouveau mot à l'utilisateur
- Affichez la liste finale.

```
PS C:\Users\kadam\Dropbox\My PC (DESKTOP-B4KCKJH)\Documents\Interface3\Python> |
```


SUPPRIMER UN ÉLÉMENT: REMOVE()

La première façon de supprimer un élément d'une liste est de dire à Python ce que vous voulez retirer via `remove()`. Python va retirer le premier élément de la liste ayant la valeur que vous spécifiez.

```
words = ["I", "really", "love", "Python"]  
words.remove("really")  
print(words)
```

Affichera :

```
["I", "love", "Python"]
```

SUPPRIMER UN ÉLÉMENT: REMOVE()

Attention ! `remove()` ne va enlever que la première occurrence de l'élément dans la liste ! Si d'autres copies se trouvent dans la liste, Python ne va pas y toucher. Dans l'exemple suivant, les deux éléments avec la valeur 1 sont colorés pour indiquer celui qui va être retiré, et celui qui va rester dans la liste:

```
numbers = [0, 1, 2, 1, 3]
numbers.remove(1)
print(numbers)
```

Affichera :

```
[0, 2, 1, 3]
```

EXERCICE

- Créez une liste contenant 10 nombres de 1 à 5.
- Affichez la liste.
- Demandez à l'utilisateur d'entrer un nombre au clavier.
- Retirez le nombre entré de la liste.
- Affichez la liste finale.

Note: Que se passe-t-il quand vous essayez de retirer un nombre qui n'est pas dans la liste ?

```
PS C:\Users\kadom\Dropbox\My PC (DESKTOP-B4KCKJH)\Documents\Interface3\Python> █
```

SUPPRIMER UN ÉLÉMENT: POP()

Parfois, plutôt que de savoir quoi enlever de la liste, vous saurez à quel index se trouve l'élément que vous voulez retirer. Ici, la méthode `pop()` va vous aider: elle retire de la liste l'élément à l'index spécifié, **et vous renvoie sa valeur**.

```
words = ["I", "really", "love", "Python"]  
removed = words.pop(1)  
print(removed)  
print(words)
```

Résultat: `really`
`["I", "love", "Python"]`

EXERCICE

- Créez une liste contenant 5 nombres de 1 à 5.
- Affichez la liste.
- Tant que la liste n'est pas vide (ex: tant que sa longueur est > 0):
 - Retirez le premier élément de la liste
 - Affichez l'élément retiré de la liste
 - Affichez la liste

```
PS C:\Users\kadam\Dropbox\My PC (DESKTOP-B4KCKJH)\Documents\Interface3\Python> 
```

MASTERMIND

"Générer le code: V1.0"

OBJECTIF

Nous allons écrire une première version de la génération du code de notre Mastermind.

Pour l'instant, vous avez une liste contenant les lettres possibles :

```
letters = ["a", "b", "c", "d", "e", "f"]
```

Nous allons maintenant essayer de construire un code de 4 lettres choisies au hasard, et obtenir une variable **code** qui contiendra une liste, par exemple :

```
["f", "c", "c", "a"]
```

MARCHE À SUIVRE

Nous allons utiliser plusieurs notions que nous avons vues :

- Création d'une liste vide dans une variable **code**
- Boucle **while** tant que quatre lettres n'ont pas été choisies:
 - Sélection d'un index au hasard de la liste de lettres à l'aide de **randint()**
 - Ajout de la lettre choisie dans la liste **code**
- Affichage du code final (pour tester)

Notes: - On peut donc bien choisir plusieurs fois la même lettre, c'est normal !
- Essayez de rendre votre code configurable, en stockant par exemple la longueur du code dans une variable, ou en utilisant la longueur de la liste de lettres dans le **randint()**.

MARCHE À SUIVRE

Nous allons utiliser plusieurs notions que nous avons vues :

- Création d'une liste vide dans une variable **code**
- Boucle **while** tant que quatre lettres n'ont pas été choisies:
 - Sélection d'un index au hasard de la liste de lettres à l'aide de **randint()**
 - Ajout de la lettre choisie dans la liste **code**
- Affichage du code final (pour tester)

```
PS C:\Users\kadom\Dropbox\My PC (DESKTOP-B4KCKJH)\Documents\Interface3\Python> █
```

RANGE

"1... 2... 3..."

UN CAS PARTICULIER

Un cas que nous allons rencontrer très régulièrement consiste à générer une séquence de nombre entiers.

- Pour générer rapidement une séquence contenant un nombre d'éléments précis
- Pour générer une liste d'index successifs
- ...

La fonction Python à utiliser dans ce cas est **range()**.

Elle génère donc une séquence de nombre entiers.

SÉQUENCES SIMPLES

`range()` va générer une suite de nombre entiers, mais pas directement une liste.

La version la plus simple consiste à créer une séquence allant de 0 à un nombre donné (non-inclus). Par exemple:

`range(10)`

va générer la séquence **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**

Python ne produit pas directement une liste par soucis d'efficacité. Imaginez que vous fassiez **`range(1000000)`** ... A la place, Python génère un objet "range" qui est capable de générer les nombres de la séquence au fur et à mesure.

RANGE → LISTE

Vous pouvez cependant forcer Python à convertir ces objets "range" en listes avec une conversion (casting) :

```
my_sequence = range(10)
my_list = list(my_sequence)
print(my_list)
```

Ou encore, en condensant (gare aux parenthèses) :

```
print(list(range(10)))
```

D'AUTRES INFOS ?

Vous pouvez également choisir le début et la fin de la séquence :

`range(2, 7)` \rightarrow 2, 3, 4, 5, 6

Ou encore, donner le "pas", ou écart, entre deux éléments successifs de la séquence:

`range(0, 10, 2)` \rightarrow 0, 2, 4, 6, 8

Exercice bonus : comment faire pour générer une liste de 10 à 1 ?

BOUCLES FOR

Le réveil de la ***force***

CHOSE PROMISE...

...chose due !

Jusqu'ici, nous avons utilisé un seul type de boucle, les boucles "**while**".

Celles-ci s'exécutent tant qu'une condition est remplie. Mais il existe un autre type de boucle, appelé "boucle arithmétique" (par opposition à "boucle logique").

Ou encore, boucle "**for**".

POURQUOI UN SECOND TYPE DE BOUCLE ?

La boucle `for` va être utile dans deux cas courants :

1. Quand vous savez combien de fois vous voulez exécuter la boucle
2. Quand vous voulez effectuer une action pour chaque élément d'une liste ou d'une autre séquence d'éléments

A noter, vous pourrez techniquement vous en sortir avec une boucle **`while`** dans tous les cas, mais la boucle **`for`** va permettre d'aller plus vite, et d'écrire du code plus clair !

ANATOMIE D'UNE BOUCLE FOR

En Python, pour utiliser une boucle **for**, il faut deux éléments :

1. Une séquence, telle qu'une liste ou un range, que la boucle va parcourir
2. Une variable, déclarée spécifiquement pour la boucle, qui va contenir chaque élément successif de la séquence.

Voici un exemple simple :

```
for i in range(10):  
    print('Compteur : ' + str(i))
```

DONC, IL SE PASSE QUOI LÀ EXACTEMENT ?

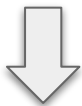
```
for i in range(10):  
    print('Compteur : ' + str(i))
```

La boucle for va parcourir les éléments d'une *séquence* (ici, `range(10)`), et va exécuter le code autant de fois qu'il y a d'éléments dans la séquence.

A chaque fois, l'élément de la séquence est mis dans une *variable* spéciale, ici `i`, qu'on peut utiliser à l'intérieur de la boucle. (Sa valeur change donc à chaque passage dans la boucle !)

PAS À PAS

```
for i in range(10):  
    print('Compteur : ' + str(i))
```



Compteur : 0

PAS À PAS

```
for i in range(10):  
    print('Compteur : ' + str(i))
```

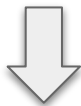


Compteur : 0

Compteur : 1

PAS À PAS

```
for i in range(10):  
    print('Compteur : ' + str(i))
```



Compteur : 0
Compteur : 1
Compteur : 2

PAS À PAS

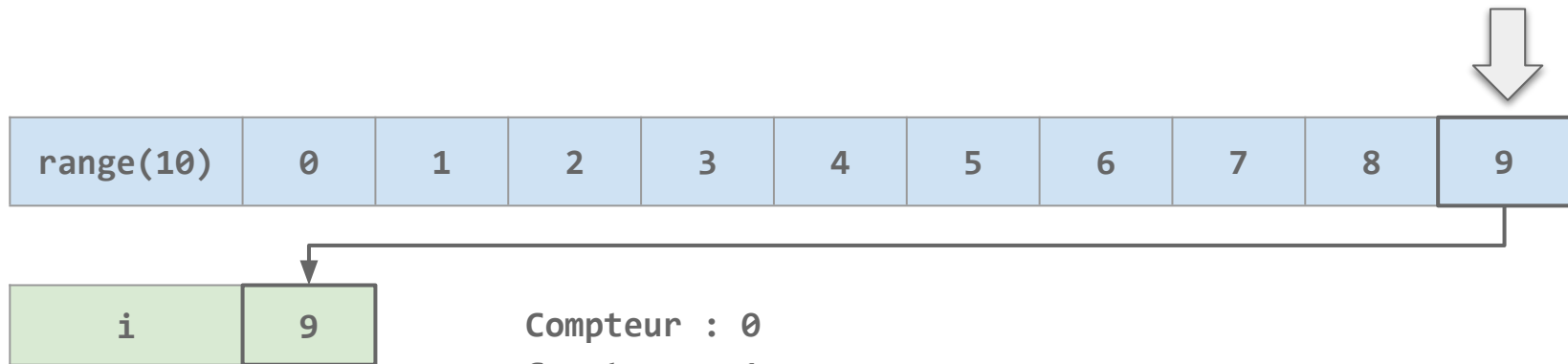
```
for i in range(10):  
    print('Compteur : ' + str(i))
```



Compteur : 0
Compteur : 1
Compteur : 2
Compteur : 3

PAS À PAS

```
for i in range(10):  
    print('Compteur : ' + str(i))
```



Compteur : 0

Compteur : 1

Compteur : 2

Compteur : 3

...

Compteur : 9

EXERCICE (* ^ _ ^ *)

Créez une liste contenant une série de mots. Ensuite, utilisez une boucle for pour itérer sur la boucle et afficher chaque élément de la liste à la console.

Par exemple, si votre liste est : `["I", "really", "love", "Python"]`, le résultat devra être :

```
I  
really  
love  
Python
```

MASTERMIND

Générer le code: V2.0

TRANSFORMER LE CODE

Nous avons maintenant tous les éléments nécessaire à la construction du Mastermind.

Dans un premier temps, nous allons améliorer le début du code que nous avons élaboré.

Essayez de remplacer la boucle **while** par une boucle **for** !

Hint: Il n'est pas obligatoire d'utiliser la variable de la boucle **for** à l'intérieur de celle-ci !

MASTERMIND

Demander une proposition au joueur

TENTATIVES

Nous allons maintenant devoir demander à notre joueur d'essayer de deviner le code.

Pour ce faire, il faut demander d'entrer une proposition de la même longueur que le code. N'oubliez pas de valider la longueur de la proposition donnée par le joueur, et d'en redemander une si cette longueur n'est pas correcte !

Ensuite, par simplicité, vous pouvez transformer la proposition du joueur en liste de lettres en utilisant `list()`.

MASTERMIND

Vérification: lettres exactes

SUITE DES ÉVÈNEMENTS :

Souvenez-vous, nous avons décidé de découper la création du Mastermind en plusieurs étapes :

1. Générer le code à deviner
2. Demander une tentative de décodage au joueur
3. Vérifier quelles lettres proposées par le joueur sont correctes
4. Mettre le tout dans une boucle
5. Gérer les lettres mal placées
6. Gérer le nombre de tentatives et afficher un message de victoire/défaite

CONFIRMER LES LETTRES EXACTES

Avant de pouvoir confirmer quelles lettres de la proposition de code du joueur sont correctes, il nous faut transformer notre chaîne de caractères en liste.

Python considère les chaînes de caractères comme des séquences de caractères (logique !), et donc vous pouvez utiliser les index, boucler dessus avec un for... Mais aussi, transformer une chaîne de caractères en liste :

```
answer = "abcd"
```

```
answer = list(answer)           →      ["a", "b", "c", "d"]
```


CONFIRMER LES LETTRES EXACTES

Une fois la réponse du joueur transformée en liste, nous allons faire une boucle for pour voir si, au même index, les éléments de la réponse du joueur et du code sont identiques. Si oui, vous pouvez mettre l'index en question dans une liste (nous en aurons besoin pour l'étape suivante).

- Comment faire pour parcourir tous les index des deux listes en parallèle ?
Difficile si on boucle sur l'une des deux listes...
- Comment afficher le nombre de lettres correctement placées après la vérification ?

MASTERMIND

Try again!

FAIRE TOUT ÇA EN BOUCLE !

L'étape suivante consiste à simplement mettre le code existant dans une boucle qui s'exécute tant que la proposition du joueur n'est pas totalement correcte.

Attention : ne pas encore s'inquiéter du nombre d'essais, nous nous occuperons de ça dans une étape suivante !

- Quel est le critère qui permet de savoir si la solution a été trouvée ?

MASTERMIND

Lettres mal placées

LA PARTIE LA PLUS COMPLEXE...

Une fois les lettres correctement placées identifiées, il est encore nécessaire de détecter les lettres qui sont correctes, mais mal placées.

Pour ça, nous allons parcourir la liste des lettres proposées par le joueur, et pour chacune :

- Vérifier chaque lettre du code
 - Si la lettre du code correspond, et que ce n'est pas une lettre dont nous avons déjà tenu compte auparavant (parce qu'elle était correcte ou parce qu'elle a déjà été identifiée comme mal placée), on stocke son index et on arrête de considérer le reste du code.
 - Sinon, on ne fait rien, et on passe à la lettre du code suivante

EXEMPLE

Si notre code est `["a", "a", "c", "d"]`
Et la proposition du joueur `["a", "e", "a", "f"]`

La liste des index corrects sera `[0]`

Nous devrions construire une liste d'index "mal placés" qui devrait contenir, à la fin:
`[1]`

Le "a" à l'index 1 du code est "mal placé" dans la proposition du joueur.
Nous aurons besoin de cet index pour s'assurer de ne pas compter plusieurs fois la même lettre mal placée.

DOUBLE BOUCLE

Nous allons devoir utiliser un cas courant: une boucle dans une autre.

Pour chaque index de la proposition du joueur, nous allons regarder chaque index du code. En première réflexion, le raisonnement est:

- Si l'un des deux index est dans notre liste d'index corrects, la lettre a déjà été "traitée", et il faut ignorer la combinaison.
- Si l'index du code correspond à une lettre mal placée identifiée auparavant, il faut aussi ignorer cet index.
- Sinon, si la lettre du code et de la proposition sont égales, il faut s'arrêter de chercher, et stocker l'index du code comme "mal placé".

DOUBLE BOUCLE

Pour écrire le code simplement, il est utile de formuler nos conditions de façon à isoler le cas où nous allons devoir prendre une action :

- Si les lettres aux deux index sont identiques:
 - Si l'index du code n'est pas dans les index corrects
 - ET l'index de la proposition n'est pas dans les index corrects
 - ET l'index du code n'est pas déjà dans les index mal placés
 - On s'arrête (et on passe à l'index suivant dans la proposition du joueur) et on ajoute l'index du code dans les index mal placés

Nous avons besoin de quelques éléments en plus pour écrire ces conditions, mais la logique est la même que plus haut, simplement formulée différemment.

IN

Pour écrire notre condition, nous avons besoin de tester si un élément (ici, un index) est présent dans une liste.

Pour ce cas précis, Python propose un petit opérateur, le **in**.

Si vous avez le code suivant:

```
words = ["I", "really", "love", "Python"]  
print("Python" in words)
```

Python va afficher **"True"** (si vous changez le mot par "web" par exemple, le code affichera **"False"** à la place.

IN

`elem in group` va donc renvoyer **True** ou **False** selon que `elem` se trouve ou pas dans `group`.

Vous pouvez l'utiliser avec des string, des listes, ... Et c'est souvent utilisé pour des conditions (attention toutefois à ne pas confondre avec le "in" présent dans une boucle `for`):

```
favorite = "Python"
if favorite in words:
    print("You have good taste in programming languages !")
```

BREAK

Enfin, une fois qu'un élément mal placé a été identifié, il n'est plus nécessaire de continuer à regarder le reste du code, et on peut passer à l'élément suivant de la proposition du joueur.

En termes de code, cette opération consiste à interrompre la boucle intérieure (celle qui itère sur les index du code) pour pouvoir passer à l'itération suivante de la boucle externe (celle qui itère sur les lettres de la proposition du joueur).

En Python, nous pouvons stopper une boucle en utilisant le mot "**break**".

BREAK

Ce mot-clé va interrompre immédiatement la boucle en cours d'exécution.

```
for index in range(20):  
    if index >= 10:  
        break  
    print(index)
```

Cette boucle ne va jamais afficher une valeur au-delà de 9, car si la valeur de **index** est **>= 10**, Python exécute un **break** et termine la boucle.

Attention, il ne s'agit pas d'une fonction (donc pas de parenthèses !).

DOUBLE BOUCLE

Armés de toutes ces notions, nous pouvons maintenant terminer le code de vérification :

- Si les lettres aux deux index sont identiques:
 - Si l'index du code n'est pas dans les index corrects
 - ET l'index de la proposition n'est pas dans les index corrects
 - ET l'index du code n'est pas déjà dans les index mal placés
 - On ajoute l'index du code dans les index mal placés et on s'arrête (et on passe à l'index suivant dans la proposition du joueur)

MASTERMIND

Nombre d'essais

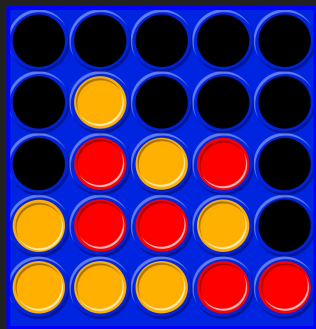
DERNIÈRE ÉTAPE !

Il reste maintenant à ajouter une condition à notre boucle, qui fait en sorte de ne laisser qu'un certain nombre de chances au joueur !

Le résultat final devrait donc ressembler à :



PUISSANCE-4-3



PUISSANCE 3

Le Puissance 3 est une variante du Puissance 4 où il faut aligner 3 jetons au lieu de 4 (horizontalement, verticalement ou en diagonale).

Cette version du jeu se joue sur une grille de 5x5 contrairement à l'original qui se joue sur une grille de 7x6.

COMMENT JOUER

Nous allons programmer la version pour deux joueurs humains (pas de joueur contrôlé par l'ordinateur donc).

- Chaque joueur va choisir à tour de rôle une colonne pour mettre son jeton. Le jeton va "tomber" jusqu'à la case vide la plus basse de cette colonne.
- Ensuite le jeu vérifiera si il y a un alignement de 3 qui s'est créé. Si oui, le joueur à gagné, sinon c'est le tour du joueur suivant.
- Si, après un coup non-gagnant, le tableau est rempli, il est vidé avant de passer au joueur suivant.

IMAGE, LONG DISCOURS, TOUT ÇA.

```
PS C:\gamedevs\tp-jam-2\TP Jam 2
puissance3.py
0 . . . . .

1 . . . . .

2 . . . . .

3 . . . . .

4 . . . . .

    0 1 2 3 4
Joueur 1(X) , quelle colonne? █
```

LISTE DE LISTES

List... en



INTO THE GRID

Pour le Puissance 3, nous allons avoir besoin de créer une grille ou un tableau.
Pour représenter cela, nous allons utiliser une liste de liste.

Pour illustrer, décortiquons le code qui crée une grille de 3 x 3 qui contient les 9 premières lettres de l'alphabet :

"a"	"b"	"c"
"d"	"e"	"f"
"g"	"h"	"i"

```
grid = [ ["a", "b", "c"], ["d", "e", "f"], ["g", "h", "i"] ]
```

INTO THE GRID

Pour vous aider vous avez le droit de la représenter cette liste comme ceci:

```
grid = [ ["a", "b", "c"],  
          ["d", "e", "f"],  
          ["g", "h", "i"] ]
```

Car, entre deux crochets (ou parenthèses), vous avez le droit de passer à la ligne.

INTO THE GRID

Alors, comme ça marche ?

Vous êtes d'accord pour dire qu'un tableau est un ensemble de lignes (ou de colonnes). Right?

Et qu'une ligne est un ensemble de cases. Toujours OK ?

Donc une ligne peut être représentée par une liste.

Par exemple, la première ligne du tableau pourrait être représentée comme suit:

```
line_1 = ["a", "b", "c"]
```

Jusque là, rien de neuf, sous le soleil.

INTO THE GRID

Faisons pareil avec les deux autres lignes:

```
line_2 = ["d", "e", "f"]
```

```
line_3 = ["g", "h", "i"]
```

Maintenant, vous pouvez considérer que votre grille est une liste de ces lignes:

```
grid = [line_1, line_2, line_3]
```

Ce qui revient à écrire:

```
grid = [["a", "b", "c"], ["d", "e", "f"], ["g", "h", "i"]]
```

CQFD :D

INTO THE GRID

On peut donc imaginer que si l'on veut récupérer une ligne du tableau on va nommer son index.

0	"a"	"b"	"c"
1	"d"	"e"	"f"
2	"g"	"h"	"i"

Par exemple:

`print(grid[0])` affichera ["a", "b", "c"]

INTO THE GRID

Une fois qu'on a une ligne, on peut accéder à l'un des éléments de cette liste via son index.

	0	1	2
0	"a"	"b"	"c"
1	"d"	"e"	"f"
2	"g"	"h"	"i"

Par exemple:

`print(grid[0][1])` affichera "b"

INTO THE GRID

Enfin, pour parcourir un tel tableau, il faudra utiliser deux boucles **for**, une pour chaque axe:

```
for line in range(3):  
    txt = ""  
    for column in range(3):  
        txt = txt + grid[line][column]  
    print(txt + "\n")
```

NB: "\n" représente un passage à la ligne.

`print("a\nb")` s'affichera comme suit:

a
b

Le code va ainsi parcourir chaque élément du tableau de cette manière:

abc
def
ghi

EXERCICES ^_^

Créer un script qui stocke un tableau avec les 12 premiers nombres entiers dans un tableau de 3x4 comme suit:

1	2	3	4
5	6	7	8
9	10	11	12

Le script affichera ce tableau, mais en divisant par deux la valeur contenue dans chaque case et en ajoutant un espace entre chaque valeur.



LES FONCTIONS

(Warning Blague Star Wars incoming)

FONCTION 101

Une fonction est un outil qui permet d'abstraire une partie du code afin d'augmenter la clarté et la lisibilité de celui-ci.

Nous avons déjà rencontré des fonctions précédemment: **len**, **print**, **input**, etc.

Ces fonctions sont mise à votre disposition par Python, mais, dans les slides suivants, nous allons voir comment créer nos propres fonctions.

COMMENT ÇA MARCHE

```
def say_hello():  
    print("Hello")
```

Voici le code déclarant la fonction **say_hello**. Nous allons revenir dans un instant sur ce code.

Maintenant qu'elle a été déclarée, nous pouvons l'utiliser dans notre code comme tout autre fonction

```
a = 3  
b = 5  
say_hello()  
print(a + b)
```

COMMENT ÇA MARCHE


Lorsqu'une fonction est appelée, c'est à dire quand elle est utilisée dans un script, le code de celle-ci est alors exécuté (dans notre exemple le `print("Hello")`)

```
a = 3
```

```
b = 5
```

```
say_hello()
```

```
print(a + b)
```



```
def say_hello():  
    print("Hello")
```


RECETTE POUR UNE BONNE FONCTION

Pour écrire une fonction il faut:

```
def say_hello():  
    print("Hello")
```

- Commencer par le mot **def** qui indique à Python votre intention d'écrire une fonction
- le nom de votre fonction, ici, **say_hello**
- Une paire de parenthèses (nous verrons plus tard qu'elles peuvent parfois être remplies) **suivie d'un ":"**
- Le code de votre fonction **indenté** (comme pour un **if** ou une boucle)

RECETTE POUR UNE BONNE FONCTION

Pour appeler une fonction il faut:

- Ecrire le nom de la fonction
- Ecrire les parenthèses (encore une fois nous verrons plus tard qu'elles ne resteront pas toujours vides)

```
a = 3  
b = 5  
say_hello()  
print(a + b)
```

Ceci est par ailleurs très similaire à **len**, **input** ou **print**.



Les noms de vos fonctions doivent **TOUJOURS** être **EXPLICITES**.

Ceci afin qu'on ne doive pas lire l'intégralité du code de la fonction pour connaître son utilité.

Maintenant que ceci est dit...

EXERCICES ^^

Créez une fonction qui affiche de manière aléatoire "**Bonjour**", "**Hello**", "**Ola**", ou "**Ciao**".

Ensuite, appelez 100 fois cette fonction.



LE RETOUR

```
def the_return_of_the_jedi():  
    the_jedi = "Ahsoka Tano"  
    return the_jedi
```

LE RETOUR...

Le retour d'une fonction est la valeur que renvoie la fonction au code appelant, par exemple, dans le cas de **input**, la valeur retournée est ce que l'utilisateur à tapé au clavier, dans le cas de **len**, c'est la longueur de la séquence que vous lui avez demandé de mesurer.

```
def dice():  
    result = randint(1,6)  
    return result
```

Dans le cas de la fonction **dice** ce sera un nombre entier entre 1 et 6.

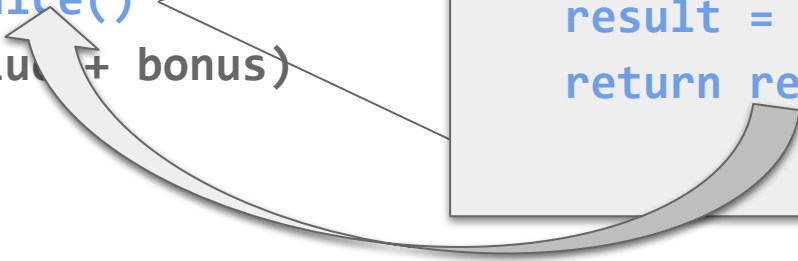
LE RETOUR...

C'est le mot-clé **return** qui indique à Python la valeur qui est retournée par la fonction. Une fois la fonction exécutée et la valeur retournée, celle-ci est substituée à l'appel de la fonction.

```
bonus = 1  
value = dice()  
print(value + bonus)
```

```
def dice():  
    result = randint(1,6)  
    return result
```

imaginons
que
randint
renvoie la
valeur 3



LE RETOUR...

Dans notre exemple la valeur retournée par **dice** est 3, donc pour le programme c'est comme si **dice**, pour cette fois-ci, valait 3.

```
bonus = 1
```

```
value = dice(→) 3
```

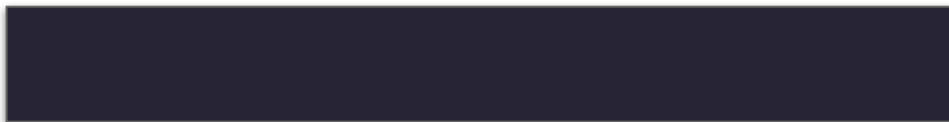
```
print(value + bonus)
```

Et donc le programme affichera 4 (3 + 1).

EXERCICES - _ -

Créez une fonction qui renvoie une lettre choisie au hasard parmi "a", "b", "c", "d" et "e".

Dans le programme principal, faites appel à cette fonction pour écrire un mot de 5 lettres.



LE RETURN INTERROMPT L'EXÉCUTION DE LA FONCTION...

Lorsqu'une valeur est retournée par une fonction, l'exécution de celle-ci s'arrête.

```
def dice():  
    result = randint(1,6)  
    return result  
    print(result)
```

Dans ce cas, le `print(result)` ne sera jamais atteint, car le code de la fonction arrêtera d'être exécuté après le `return`.

DEUX RETOURS?

Une fonction peut pertinemment contenir cependant deux retours si leurs exécution est conditionnées.

```
def test():  
    result = randint(1,6)  
    if result < 4:  
        return True  
    return False
```

Quand cette fonction est exécutée, si la valeur de result est plus petit que 4, elle retournera True, sinon elle retournera False.

ENTRE PARENTHÈSE

PARAMÈTRE...

Toute fonction peut avoir un ou plusieurs paramètres.

```
from random import randint
```

```
dice = randint(1, 6)
```

Dans cet exemple, 1 et 6 sont les paramètres de la fonction **randint**.

...ARGUMENT

Pour pouvoir utiliser des paramètres, une fonction a besoin d'arguments

```
def double(number):  
    return number * 2
```

Ici **number** est l'argument de la fonction **double**.

Les paramètres sont donc les valeurs passées à la fonction lors de son appel.

Alors que les arguments sont les "variables" déclarées dans l'en-tête de la fonction et qui serviront à accueillir les valeurs des paramètres.

```
def double(number):  
    return number * 2
```

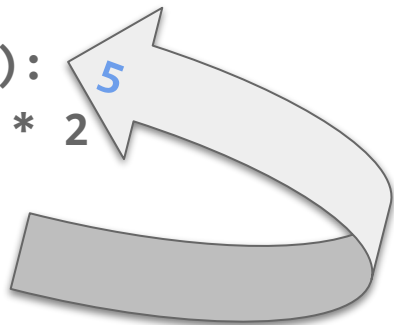
```
value = double(5)  
print(value)
```

ARGUMENT... PARAMÈTRES ...?

Lors de l'appel d'une fonction, le compilateur passe la valeur des paramètres dans les arguments respectifs:

```
def double(number):  
    return number * 2
```

```
value = double(5)  
print(value)
```



Ce qui veut dire que pour cet appel de la fonction **double**, **number** vaudra **5** (et donc la valeur retournée sera **10**)

EXERCICES > _ <

Créez une fonction qui prendra en paramètre un nombre de lettres désiré et qui renverra un mot de la longueur correspondante composé de lettres prises au hasard parmi "a", "h", "k", "o", "n", "s" et "t".

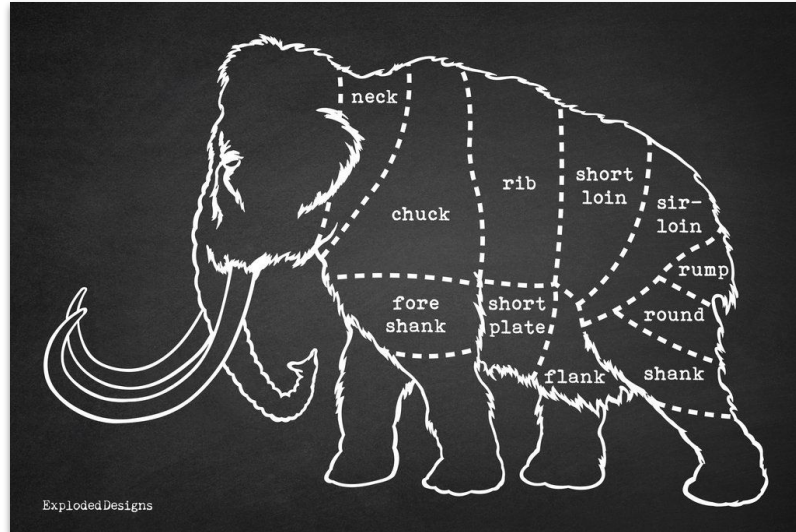
Le programme principal demandera à l'utilisateur combien de lettres il veut dans son mot et affichera ensuite un mot fabriqué par la fonction.

```
on.exe d:/IF3/20210111/test.py  
Combien de lettre: █
```

PUISSANCE 3

DIVIDE AND CONQUER

Normalement, vous avez maintenant tout le bagage pour faire le jeu de A à Z, mais certainement encore un peu de mal à imaginer par quel bout prendre ce problème. Donc, nous allons voir comment "découper le mammoth".



CUTTING THE MAMMOTH

La première étape face à ce genre de problème est d'établir la logique générale. Ici, par exemple, nous pouvons découper le programme en trois phases.

- Initialisation
- Jeu
- Déclaration de la victoire

PUISSANCE 3

Initialisation

CONTINUE TO CUT THE MAMMOTH

Maintenant, faisons le même exercice avec chaque phase.

Cette phase-ci ne demande pas grand chose:

- il faut initialiser le tableau 5X5 en le remplissant avec des "." (pour représenter les espaces vide).
- Une fois que c'est fait, nous allons créer une fonction capable d'afficher le tableau.

CONTINUE TO CUT THE MAMMOTH

La fonction doit prendre en paramètre un tableau et gérer l'affichage afin de faire en sorte d'afficher le tableau avec des numéros indiquant les lignes et les colonnes et ajoutant un espace entre chaque colonne.

L'affichage devrait ressembler à ça:

0
1
2
3
4
	0	1	2	3	4

PUISSANCE 3

Le jeu

ANOTHER SLICE

Encore une fois, nous allons pouvoir découper cette étape en plusieurs parties.

Premièrement, le jeu continue tant que personne n'a gagné, donc avant toute chose on va déclarer une variable qui contiendra le vainqueur, celle-ci sera bien évidemment null au départ.

Notre jeu continuera tant que cette variable reste null, mais on reviendra dessus dans un instant.

ANOTHER SLICE

Ensuite, il nous faut aussi une variable pour connaître le joueur qui doit jouer, afin d'assurer l'alternance des tours.

Et donc après toutes ces déclarations, on va pouvoir écrire la boucle logique qui nous permettra de jouer tant qu'il n'y a pas de vainqueur (et donc tant que la variable dédiée au vainqueur est null).

À chaque itération de la boucle, exécutera un tour de jeu.

ANOTHER SLICE

Un tour de jeu se déroule comme suit:

- joueur joue
- on checke si le joueur a gagné
 - si le joueur à gagné on s'arrête
 - sinon on passe au joueur suivant

On va donc créer une fonction qui prend en paramètre le tableau et le joueur courant (le joueur sera symbolisé par un entier (**1** pour "joueur 1" et **2** pour "joueur 2")).

Cette fonction renverra **True** si le joueur passé en paramètre a gagné et **False** dans le cas contraire.

ANOTHER SLICE

Dans un premier temps faites en sorte qu'elle renvoie **True** si le joueur est 2 (c'est en attendant de développer plus le code).

Dans le code principal, faites en sorte que le jeu continue tant que l'un des joueurs n'a pas gagné en prenant soin de suivre les étapes décrites dans le slide précédent.

PUISSANCE 3

La déclaration du gagnant

LAST SLICE

Et donc, une fois la partie terminée, il suffit d'annoncer le gagnant et afficher une dernière fois le tableau (afin d'attester de la victoire de celui-ci).

Normalement dans l'état du code actuel, le joueur 2 devrait gagner à chaque coup et le tableau devrait rester vide. Mais patience, nous avançons pas à pas.

PUISSANCE 3

Inside the game

CUT THE SLICE INTO PIECES

Nous allons créer une fonction qui prend en paramètre un tableau, un joueur et une colonne (un entier allant de 0 à 4 dans notre cas).

Cette fonction renverra le tableau qu'elle a reçu après y avoir placé un "jeton", dans la colonne passée en paramètre.

Le jeton sera symbolisé par "O" pour le "joueur 1" et par "X" pour le "joueur 2".

Placez le jeton dans la colonne indiquée, dans la position la plus basse possible. Si il n'y a pas de place dans la colonne, rien ne se passe (et donc le tour est perdu).

CUT THE SLICE INTO PIECES

Dans la fonction qui symbolise le tour de jeu:

- Affichez le tableau (rappelez vous, il y a une fonction pour ça)
- Demandez à l'utilisateur dans quelle colonne il désire jouer (attention, ça doit être un entier)
- Utilisez la nouvelle fonction pour ajouter un jeton dans le tableau

```
PS C:\game devs\tp-jam-2\TP Jam 2
puissance3-1.py
0 . . . . .
1 . . . . .
2 . . . . .
3 . . . . .
4 . . . . .

  0 1 2 3 4
Joueur 1(X) , quelle colonne? █
```

PUISSANCE 3

Checks

VÉRIFIER SI C'EST GAGNÉ

Après chaque coup joué, vous devez vérifier que ce coup n'a pas fait gagner le joueur.

Encore une fois, c'est une fonction qui va s'occuper de ça.

Et cette fonction prendra en paramètres le tableau et la colonne dans laquelle le coup a été joué.

La fonction déterminera la ligne où a atterri le jeton (c'est la première ligne non-vide de la colonne) et connaîtra ainsi le jeton qui a été joué (X ou O).

Ensuite, avec ces informations, la fonction va vérifier les trois possibilités de victoire: une ligne horizontale, verticale ou en diagonale.

Chacune de ces vérifications fera l'objet d'une fonction à part entière.

VERTICALE

Pour la ligne verticale, il y n'a qu'une possibilité qui amène une victoire: les deux jetons en dessous du jeton joué son identiques à celui-ci:

0
1	.	0	.	.	.
2	.	0	.	.	.
3	.	0	X	.	.
4		X	X	0	.
	0	1	2	3	4

Créez donc une fonction qui renvoie **True** si c'est le cas et **False** sinon.

HORIZONTALE

Pour la ligne horizontale, il y a 3 possibilités.

Les jetons à droite sont identiques à celui joué.

0
1
2
3	.	X	.	.	.
4	X	X	0	0	0
	0	1	2	3	4

Les jetons à gauche sont identiques à celui joué.

0
1
2
3	X	X	.	.	.
4	0	0	0	X	.
	0	1	2	3	4

Le jeton joué est entouré de jetons identiques

0
1
2
3	X	.	X	.	.
4	0	0	0	.	.
	0	1	2	3	4

Créez donc une fonction qui renvoie **True** dans ces cas et **False** dans tous les autres cas.

DIAGONALE

Pour la ligne diagonale, il y a 6 possibilités.

0
1
2	X	0	.	.	.
3	0	X	.	.	0
4	X	X	X	0	0
	0	1	2	3	4

0
1
2	X	0	.	0	.
3	X	X	.	X	X
4	0	0	X	X	0
	0	1	2	3	4

0
1
2	X	.	.	X	.
3	X	X	.	0	.
4	0	0	X	0	.
	0	1	2	3	4

DIAGONALE

0
1
2	.	.	X	.	.
3	.	X	0	.	.
4	X	X	X	0	0
	0	1	2	3	4

0
1
2	.	.	0	.	.
3	X	0	X	.	.
4	0	0	X	.	.
	0	1	2	3	4

0
1
2	.	.	0	.	.
3	X	0	X	.	.
4	0	X	0	.	.
	0	1	2	3	4

Créez donc une fonction qui renvoie **True** dans ces cas et **False** dans tous les autres cas

VÉRIFIER SI C'EST GAGNÉ

Une fois que vous avez vos trois fonctions de vérification, vous pouvez compléter la fonction de vérification globale qui va faire appel à chacune d'entre elle.

Et une fois que vous avez cette fonction, elle va correctement vérifier si le joueur qui vient de jouer a gagné.

Une fois que c'est fait, vous avez plus ou moins votre Puissance 3. Il reste une dernière étape.

PUISSANCE 3

Last step

QUAND C'EST REMPLI

Il ne reste plus qu'à vérifier que lorsqu'un coup a été joué, si il n'est pas gagnant, il ne remplit pas la dernière case du tableau.

Pour ça, écrivez une fonction qui prend un tableau en paramètre et qui retourne **True** si celui-ci est rempli, ou **False** dans le cas contraire.

Tip: le tableau est rempli si la ligne 0 du tableau est remplie

Une fois que vous avez cette fonction, il ne vous reste qu'à la placer dans votre programme principal et le tour est joué.

RESULTAT FINAL

Le résultat final devrait ressembler à ceci:

```
PS C:\game devs\tp-jam-2\TP Jam 2
puissance3.py
0 . . . . .

1 . . . . .

2 . . . . .

3 . . . . .

4 . . . . .

    0 1 2 3 4
Joueur 1(X) , quelle colonne? █
```