Découverte de Python

GCC ! – Prologin

2020

Alice se réjouit du tournoi de sport qui s'organise en ce moment. Seulement, Alice déteste une chose : compter les points. Alice décide donc qu'elle va apprendre à se servir du langage informatique Python, pour que son ordinateur garde le compte à sa place.

Ce TP va suivre sa progression. Il y aura beaucoup d'exemples et il ne faut pas hésiter à les tester soi-même ! Faire des sauvegardes régulières et créer plusieurs fichiers distincts sont aussi de bonnes habitudes à avoir !

Afficher le score dans le terminal

En premier, elle apprend que si elle écrit un programme `score.py`, il ne s'exécute que si elle rentre la commande `python3 score.py` dans le terminal depuis le dossier où se trouve son fichier `score.py`.

Au passage, l'extension `.py` sert à préciser à l'ordinateur que ce qui est écrit dans le fichier est écrit dans le langage python, ce qui peut permettre à certains éditeurs de colorer les mots clefs de Python.

Ensuite elle veut voir si elle peut afficher un texte ou un nombre en sortie dans le terminal. Elle découvre qu'il existe une fonction `print` qui peut lui permettre d'afficher plein de choses dans le terminal quand elle exécute le programme.

Contenu du fichier `score.py` :

```
1 print("Bonjour !")
```

Résultat à l'exécution dans la console :

```
Bonjour !
```

Elle remarque ensuite qu'en écrivant plusieurs lignes à la suite, elles s'affichent dans le même ordre dans le terminal. Et elle apprend également que si elle commence une ligne par un `#` la ligne n'est pas prise en compte. On s'en sert généralement pour laisser des commentaires dans son code :

```python
print("Bonjour,")
# print("Ceci ne va pas être affiché")
# C'est juste pour l'exemple !
print("Bienvenue au stage informatique GCC! !")
```

```
Bonjour,
Bienvenue au stage informatique GCC! !
```

Mais à un moment, elle découvre que si elle oublie certains éléments, le programme ne fonctionne plus. Par exemple, si elle oublie de fermer une parenthèse, de fermer des guillemets ou si elle oublie complètement les guillemets, elle peut obtenir toutes ces erreurs :

Il manque une parenthèse :

```python
print("Score :"
```

```
SyntaxError: unexpected EOF while parsing
```

Il manque une parenthèse :

```python
print"Score :")
```

```
SyntaxError: invalid syntax
```

Il manque un guillemet :

```python
print("Score :)
```

```
SyntaxError: EOL while scanning string literal
```

Il manque des guillemets :

```python
print(Score)
```

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

Il manque des guillemets :

```python
print(Score :)
```

```
SyntaxError: invalid syntax
```

Les erreurs apparaissent lorsqu'on a fait une faute en écrivant notre programme. Il est important d'apprendre à les comprendre pour trouver les fautes et les corriger. Notamment quand il y a une erreur, il y a toujours une ligne précisée, ce qui peut te donner une idée d'où se trouve la faute.

Exercice 1 Essaie de créer un fichier `score.py` qui, quand on le lance dans la console, affiche :

```
Le score est de 3 contre 1 !
```

Garder le score en mémoire

Maintenant, Alice se dit qu'elle sera capable d'afficher les scores par écrit dans un terminal, mais elle ne veut toujours pas avoir à compter les points...

Alice apprend alors qu'il existe une commande, `input`, qui permet d'attendre et de récupérer des informations depuis le terminal. `input()` bloque l'exécution du programme jusqu'à ce que l'utilisateur écrive quelque chose dans le terminal et appuie sur la touche *entrée*. Donc par exemple, si tu fais un programme avec juste `input()` et que tu exécutes ton programme, un terminal s'ouvre. Tu peux alors écrire ce que tu veux, et il ne se passera rien tant que tu n'appuieras pas sur la touche *entrée*.

Seulement, si on écrit juste :

```
1 input()
```

On perd instantanément ce que l'utilisateur a écrit, car on n'a pas fait en sorte de le sauvegarder ! Pour le sauvegarder on peut par exemple écrire :

```
1 entree = input()
```

Ici, grâce au `=`, on écrit qu'on sauvegarde le résultat de `input()` dans la *variable* `entree`. Cela nous permettra d'avoir accès autant qu'on veut à ce que l'utilisateur a écrit. Par exemple, on peut afficher les variables, et donc écrire un programme qui recopie ce que l'utilisateur lui donne :

```
1 entree = input()
2 print(entree)
```

Alice s'est débrouillée pour n'avoir que les points d'une équipe à compter. Pour compter les points, elle va avoir besoin de variables (par exemple `score` et `points_marques`) qui sont des entiers, qu'elle pourra sommer, soustraire, multiplier ou encore diviser. Par exemple, si elle lance le programme suivant, elle obtient :

```
1 score = 1
2 points_marques = 2
3 score_total = score + points_marques
4 print(score_total)
```

```
3
```


Plus précisément, elle veut écrire un programme qui attend de l'utilisateur (on dit aussi qui prend en entrée) le nombre de points qu'a l'équipe en ce moment, puis prend en entrée le nombre de points marqués, et ensuite affiche dans le terminal (on dit aussi qu'il écrit en sortie) le nouveau total.

Pour cela, son programme va devoir récupérer des entiers. Une façon de le faire est à l'aide de la fonction `int`. Cette fonction permet de convertir ce que l'on récupère, c'est-à-dire des chaînes de caractères, en des entiers. Mais ça ne marche pas si la chaîne n'est pas un entier :

```
1 print(int("4") + 1)
```

```
5
```

```
1 int("a")
```

```
ValueError: invalid literal for int() with base 10: 'a'
```

Pour récupérer un entier, tu peux donc écrire l'instruction ci-dessous ; mais si tu ne donnes pas un entier quand tu exécutes le programme, tu auras une erreur :

```
1 nombre = int(input())
```

Exercice 2 Tu peux maintenant écrire le programme qui récupère le total des points de l'équipe et les points qu'elle a marqués, et affiche le nouveau total.

Par exemple si je lance le programme, que j'écris `5` puis `2`, le programme devra écrire `7`, car $5 + 2 = 7$. Mon terminal ressemblera alors à :

```
5
2
7
```

Alice se dit qu'elle aimerait bien avoir une présentation un peu plus jolie. Elle voudrait que ce qui s'affiche dans le terminal soit au final :

```
Le score est ?
5
Le nombre de points marqués est ?
2
Alors le score est :
7
```

`Astuce :` Si tu as stocké le score dans la variable `score` qui vaut par exemple 3, tu peux aussi faire :

```
1 print("Le score est", score)
```

```
Le score est 3
```


Essaie toi aussi de faire une jolie présentation !

Dire qui a gagné

Maintenant, Alice aimerait pouvoir écrire quelle équipe est en train de gagner. Son programme aura en entrée le score de l'équipe 1, puis le score de l'équipe 2, et il devra afficher *l'équipe 1 a gagné* ou *l'équipe 2 a gagné* en fonction de l'équipe ayant le score le plus élevé.

Pour cela, Alice va avoir besoin de comparer des nombres, et d'agir en fonction du résultat. En Python, on peut écrire des tests dont les résultats seront soit *vrai* soit *faux*, ou plus précisément soit `True`, soit `False`. Ce sont des types spéciaux : des *booléens*.

Il existe plein de façons de faire des tests : une comparaison stricte entre deux entiers, qui dit vrai si `a` est plus petit que `b` (`a < b`), une comparaison large, qui dit vrai si `a` est plus petit ou égal à `b` (`a <= b`), un test d'égalité, qui dit vrai si `a` est égal à `b` (`a == b`) ou encore un test de différence, qui dit vrai si a est différent de b (`a != b`).

Ces tests, combinés à ce qu'on appelle des structures conditionnelles, permettent à un programme, qui est toujours écrit de la même façon, de s'exécuter différemment en fonction des valeurs données par l'utilisateur. La première que nous allons découvrir est la structure conditionnelle *si* (`if` en anglais). Elle fonctionne sur le principe qu'on ne fait quelque chose que si le test est vrai.

Par exemple :

```
1 n = int(input())
2 if n < 0:
3     print("le nombre est négatif !")
```

Ce programme attend un nombre de l'utilisateur. Si ce nombre est négatif, il affiche «le nombre est négatif !». Sinon, il n'affiche rien.

Alice se demande comment faire si elle veut tester deux conditions en même temps. Par exemple, si elle se pose la question : *Un bonbon* ***ou*** *un sort ?* . Elle découvre alors qu'il est possible de faire cela en Python, grâce aux *opérateurs logiques*. Il en existe trois :

- **or** qui signifie *ou* en français
- **and** qui signifie *et*
- **not** qui signifie *non*

Si on utilise `or`, le test sera *vrai* si l'un des deux tests qui le composent ou les deux sont *vrais*. On l'utilise comme suit :

```
1 if TEST1 or TEST2:
2   # ce que je fais si l'un des deux tests ou les deux sont vrais
```

C'est exactement la même chose dans le cas du *et*. La seule différence est que pour que le test soit *vrai*, il faut que les deux tests qui le composent soient *vrais* tous les deux ! En Python, on écrit :

```
if TEST1 and TEST2:
  # ce que je fais si les deux tests sont vrais
```

Dans la syntaxe (la façon dont on écrit notre programme), il y a deux choses très importantes pour que le programme s'exécute correctement : les `:` après le test du `if`, et le décalage des lignes suivantes. Ce décalage s'appelle *l'indentation*. Il permet de distinguer ce qui sera fait de la suite du programme, si le test est vrai. On le crée avec la touche *tab* ou des espaces.

Cette structure peut avoir d'autres parties : un *si* peut être suivi d'un *sinon* (`else`) et ce qui suit le *sinon* ne sera exécuté que si le test du *si* est faux ! Un peu comme je pourrais dire *Si j'ai mes chaussettes, je mets mes chaussures, sinon je mets mes chaussettes.* . Et puis il y a une troisième possibilité, les *sinon si* (`elif`) : on exécute ce qui suit le *sinon si* seulement si tous les tests précédents étaient faux et celui là est vrai. Un peu comme je pourrais dire *Si j'ai mes chaussettes, je mets mes chaussures, sinon si j'ai des sandales je les mets, sinon je mets des chaussettes.* .

Avec l'indentation, le schéma reste toujours :

```
if TEST1:
    # ce que je fais si le test 1 est vrai
elif TEST2:
    # ce que je fais si le test 1 est faux et le test2 est
    # vrai
elif TEST3:
    # ce que je fais si tous les tests précédents sont faux
    # et le test 3 est vrai
# (il peut y avoir autant de elif que nécessaire)
else:
    # ce que je fais sinon

# ce que je fais après dans tous les cas
```

Par exemple, le programme suivant respecte ce schéma :

```
print("Quel est le nombre ?")
n = int(input())

if n < 0:
    print("le nombre est négatif !")
elif n == 0:
    print("le nombre est zéro !")
else:
    print("le nombre est positif !")
```

Et il donne les exécutions suivantes :

```
Quel est le nombre ?
3
le nombre est positif !
```

```
Quel est le nombre ?
-1
le nombre est négatif !
```

```
Quel est le nombre ?
0
le nombre est zéro !
```

Exercice 3 Alice et toi avez maintenant toutes les armes en main pour faire un programme qui prend en entrée deux nombres, et affiche en sortie l'équipe qui a gagné, comme sur l'exemple qui suit. Lance toi !

```
5
7
L'équipe 2 gagne !
```

Les matchs de ping pong

Alice doit maintenant arbitrer un match de ping pong. Avant le début du match, les deux joueurs se mettent d'accord sur un nombre de manches qu'ils vont jouer. C'est celui qui, à la fin, a gagné le plus de manches qui remporte le match. Alice souhaite donc maintenant indiquer à son programme qui a gagné pour chacune des manches et que le programme lui dise qui a gagné à la fin. Seulement, le nombre de manches varie d'un match à l'autre, le nombre de questions posées par l'ordinateur doit donc aussi varier. Pour cela, Alice va avoir besoin de découvrir un nouveau type de structures conditionnelles : les boucles.

Il s'agit de la boucle *pour* (`for` en anglais). Elle nous permet de dire que l'on va répéter un bout de notre code plusieurs fois, pour plusieurs valeurs. En fait, pour dire que je vais répéter quelque chose 4 fois, je vais dire que je vais le répéter pour une variable `i` prenant les valeurs `0` puis `1`, `2`, et enfin `3`.

Par exemple, si je veux afficher 4 fois le texte *Je ne veux pas compter les points du match* , je peux écrire le code :

```
1 for i in range(4):
2     print("Je ne veux pas compter les points du match")
```

```
Je ne veux pas compter les points du match
Je ne veux pas compter les points du match
Je ne veux pas compter les points du match
Je ne veux pas compter les points du match
```

Par contre le code suivant donne un résultat qui n'est pas identique les 4 fois :

```
for i in range(4):
    print("Je ne veux pas compter les points du match", i)
```

```
Je ne veux pas compter les points du match 0
Je ne veux pas compter les points du match 1
Je ne veux pas compter les points du match 2
Je ne veux pas compter les points du match 3
```

Pour un nombre `n` de fois, on va dire que la variable `i` prend les valeurs `0`, `1`, ..., `n-2` et `n-1`. Mais attention, la variable n'a pas besoin de s'appeler `i` ! Son nom n'a pas d'importance, à part pour la mentionner dans la boucle.

Si je veux répéter un nombre `n` fois un bout de code, on écrira selon le schéma suivant. Rappelle-toi, le décalage visuel (*indentation*) est important :

```
for i in range(n):
    # ce que je veux répéter n fois (où i peut intervenir)

# la suite du programme sera exécutée une seule fois
```

Exercice 4 Alice et toi êtes désormais prêtes pour écrire une fonction qui demande d'abord le nombre de manches, puis les résultats des manches, et enfin le programme annonçant le vainqueur du match :

```
Combien de manches ?
3
Qui gagne les manches ?
1
2
2
Le gagnant est 2
```

Les matchs de tennis

C'est désormais le tour des matchs de tennis dans le tournoi !

Les règles mises en place sont les suivantes : une manche continue tant qu'aucun des joueurs n'a atteint le score de 6 points, et quand un des joueurs l'atteint, il remporte la manche. Chaque match est composé de 3 manches. Mais Alice se rend compte qu'il n'est pas très facile d'écrire un programme qui prend en entrée, un par un, les points marqués par les joueurs, tant que la manche n'est pas finie avec seulement des boucles `for`. Effectivement, il existe une boucle, similaire à la boucle `for`, plus adaptée pour faire cela : la boucle `while`, qui se traduit en français par *tant que*.

On s'en servira comme cela :

```
while condition: # tant que la condition est vraie
    # ce que je veux répéter

# la suite du programme sera exécutée une seule fois
```

Par exemple, le programme suivant va demander à l'utilisateur ce que vaut 2+2 tant qu'il n'aura pas répondu 4 :

```
resultat = 0

while resultat != 4:
    resultat = int(input("Que vaut 2+2 ? "))

print("2+2 = 4")
```

Seulement, il y a un danger avec cette boucle : si on ne fait rien pour que la condition change à un moment, on reste bloqué dans la boucle indéfiniment. Il faut donc faire attention à modifier la variable de la condition, lorsqu'on emploie cette boucle.

Par exemple, *à cause d'une erreur* d'inattention ce programme ne terminera jamais car on ne change pas la variable `resultat` utilisée dans la condition :

```
resultat = 0

while resultat != 4:
    # /!\ on voulait probablement utiliser la
    # variable `resultat`  et non pas `entree`
    entree = int(input("Que vaut 2+2 ?"))

print("2+2 = 4")
```

Exercice 5 Maintenant, essaie d'écrire un programme qui demande qui a marqué le point tant que la manche n'est pas finie (personne n'a atteint 6 points) et annonce le gagnant à la fin :

```
1
1
1
2
1
2
1
2
1
Le joueur 1 remporte la manche !
```

Simplifier le code

On va désormais chercher à simplifier la gestion des manches, afin de pouvoir demander à jouer un nombre variable de manches dans un même match. Pour cela, on va utiliser un nouvel outil, les *fonctions*. Une fonction, c'est un bout de code qui prend en entrée des choses, et donne en sortie d'autres choses. En fait, ce n'est pas la première fois que l'on utilise des fonctions : `print` et `input` étaient des fonctions. Alice a aussi vu en cours de maths des fonctions , comme $f(x) = 2x + 1$, et se dit qu'on pourrait l'écrire en informatique aussi ! Ici, l'entrée est un premier nombre, et la sortie un autre nombre, égal au premier multiplié par deux, et auquel on a ajouté 1. En Python, on peut l'écrire comme cela :

```
1 def f(x):
2     return 2 * x + 1
```

Puis, après avoir défini la fonction, on peut l'utiliser. On dit qu'on *appelle* la fonction. Par exemple, l'exécution du programme suivant donnera comme résultat :

```
1 def f(x):
2     return 2 * x + 1
3
4 print(f(0))
5 print(f(1))
```

```
1
3
```

Seulement, les fonctions que l'on va utiliser pour simplifier la gestion des manches ne peuvent pas juste être des fonctions mathématiques. Elles ont besoin d'avoir accès à tous les outils qu'on a vus auparavant : les conditions `if` , les boucles `for` ou `while` , ou encore les fonctions `input` ou `print` . Finalement, les fonctions vont nous permettre en quelque sorte de définir un programme dans notre programme !

Par exemple, si on a souvent besoin d'afficher une ligne de 10 '*', on pourrait définir la fonction `dix_etoiles` ci-dessous. Par contre, nous pourrions aussi avoir besoin d'afficher souvent une

ligne d'étoiles mais avec un nombre d'étoiles qui varie. Dans ce cas là, on définit la fonction `n_etoiles`.

```
def dix_etoiles():
    for i in range(10):
        print("*", end="")

def n_etoiles(n):
    for i in range(n):
        print("*", end="")

dix_etoiles()
print() # retour à la ligne
n_etoiles(3)
print() # retour à la ligne
n_etoiles(6)
print() # retour à la ligne
```

```
**********
***
******
```

La définition d'une fonction est toujours introduite par le mot clef `def`, puis vient le nom de la fonction (sans espace). Puis, entre parenthèses, on met les objets que la fonction prend en entrée (que l'on appelle aussi *arguments/paramètres*), comme par exemple `n` dans la fonction `n_etoiles` ou encore `x` dans `f` ; il peut y en avoir plusieurs. Dans ce cas, on les énumère à l'aide de virgules. On pourra les utiliser dans la fonction, comme si on connaissait leur valeur. Puis vient le : , suivi d'un retour à la ligne et d'une indentation, comme pour les boucles. En résumé, la syntaxe des fonctions est :

```
def nom_de_ma_fonction(arg1, arg2,..., dernier_arg):
    # Le code de ma fonction
    # on peut utiliser les arguments

# Le code que mon programme exécutera directement
# et qui peut faire appel à ma fonction
```

Mais il reste une dernière chose à clarifier : quand on a défini `f`, on a utilisé le mot clef `return`. Ce mot clef nous permet de récupérer un résultat après l'appel de la fonction, alors que si j'utilise `print`, je ne fais qu'afficher le résultat dans la console, et je ne peux pas m'en servir après. Par exemple, si j'écris le code suivant j'obtiens l'erreur :

```python
def f(x):
    print(2 * x + 1)

print(f(2) + 1)
```

```
TypeError: unsupported operand type(s) for +: 'NoneType' and
'int'
```

Ce qu'il faut faire est :

```python
def f(x):
    return 2 * x + 1
print(f(2) + 1)
```

Exercice 6 Maintenant, avec tout ce que tu as appris, écris une fonction `manche` qui prend en entrée le score à atteindre, et qui renvoie le gagnant (`1` ou `2`). Ainsi le programme suivant pourra organiser un match et annoncer le gagnant !

```python
nb_manches = int(input("Combien de manches ? "))
score_j1 = 0
score_j2 = 0

for i in range(nb_manches):
    score = int(input("Quel est le score à atteindre ? "))
    gagnant = manche(score)

    if gagnant == 1:
        score_j1 += 1
    else:
        score_j2 += 1

if score_j1 > score_j2:
    print("Le joueur 1 gagne le match !")
else:
    print("Le joueur 2 gagne le match !")
```

À noter que, si je le souhaite, je peux mettre tout le code au-dessus dans une fonction `match` , afin de l'appeler plusieurs fois pour jouer plusieurs matchs à la suite, sans devoir relancer mon programme !

Fini !

Alice est fière d'elle, elle a réussi à faire ce qu'elle voulait (ne pas compter les points) et elle a découvert beaucoup d'outils très pratiques ! Maintenant, il ne reste plus qu'à les utiliser pour faire plein d'autres choses.

Si tu as fini en avance, tu peux demander plus d'exercices !