Belgium

esero

Fiche pour l'étudiant

Mission Space Lab

Phase 1 : Introduction à la programmation Python

Dans cette première phase de préparation, les élèves acquièrent les compétences minimales en programmation nécessaires pour mener à bien le défi de la Mission Space Lab: estimer la vitesse de la Station Spatiale Internationale à partir de photos prises à son bord en programmant le mini-ordinateur Astro Pi.

esa

belspo

La Scientothèque
L'ÉGALITÉ DES CHANCES PAR LES SCIENCES

ULB

Introduction à Python

Préliminaires à la Mission Space Lab

Dossier élève

Table des matières

Introduction

Python est un langage très apprécié pour la simplicité de sa syntaxe, permettant ainsi un développement rapide de code. On dit de Python qu'il est un langage interprété, par opposition aux langages compilés, car on peut exécuter directement chaque ligne de code dès qu'on l'a écrite. Sa simplicité est obtenue au prix d'une plus grande lenteur par rapport à d'autres langages comme le C ou le C++. Cependant, de nombreuses améliorations au fil des années ont rendu cette lenteur de moins en moins perceptible. Très populaire dans de nombreux domaines, il est notamment connu en Intelligence Artificielle (IA) qui a permis de créer les outils populaires comme ChatGPT.

Dans la suite, une très brève introduction aux éléments importants du langage est présentée. Ceci ne constitue en rien un cours introductif complet mais vise à permettre l'utilisation pratique des éléments utiles pour ce projet. Attention : c'est la dernière version du langage qui sera utilisé, c'est à dire Python 3 !

Un premier programme

Classiquement, le premier programme à être écrit dans un cours de programmation est celui qui consiste à afficher la célèbre phrase ***"Hello World!"***. Contrairement à bien d'autres langages, réaliser un tel programme en Python est des plus simples grâce à l'instruction `print()` qui permet d'afficher à l'écran ce qui est placé entre les parenthèses. Il suffit donc d'y écrire la phrase voulue entre guillemets :

```
print("Hello world!")
```

Les variables

Les variables permettent d'enregistrer des données. Cela peut être un nombre entier, à virgule, une lettre, un mot, une phrase et bien plus. La variable permet de garder dans la mémoire de l'ordinateur l'information voulue pour être utilisé plus tard dans le programme.

Pour créer une variable, il suffit de lui donner un nom qui ne commence pas par un chiffre et lui affecter une valeur :

```
a = 5
maVariable = 3.14
```

IMPORTANT

Contrairement aux mathématiques, le symbole `=` ne correspond pas à une égalité, c.-à-d. que l'on n'exprime pas ici *"maVariable est égal à 3.14"* dans le sens d'une comparaison mais d'une ***affectation***. Cela signifie que pour Python la valeur 3.14 est affectée à la variable `maVariable` et que cette affectation s'opère de droite à gauche. En effet, `3.14 = maVariable` n'a aucun sens pour Python car cela voudrait dire que l'on affecte la variable `maVariable` à la valeur 3.14 alors qu'en mathématique cette expression reste valable.

Pour afficher la valeur enregistrée dans la variable, on utilise la commande `print()` de la manière suivante:

```
print(a)
print(maVariable)
```

Après exécution, tu verras dans le résultat suivant

```
5
3.14
```

Pour enregistrer une lettre, un mot ou une phrase, on l'écrit *entre guillemets* ou *apostrophes* simples :

```
uneLettre = 'b'
unMot = "anticonstitutionellement"
unePhrase = "Bonjour tout le monde!"
```

Pour afficher le contenu de ces variables, il suffit d'utiliser la même commande `print()`

```
print(uneLettre)
print(unMot)
print(unePhrase)
```

qui produira le résultat suivant

```
b
anticonstitutionellement
Bonjour tout le monde!
```

L'intérêt d'avoir deux manières de représenter une phrase tient au fait que certaines contiennent déjà une apostrophe ou un guillemet. Ainsi, une variable contenant la phrase `J'ai faim` devra être mise entre guillemets puisqu'il y a déjà une apostrophe présente

```
maVariable = "J'ai faim"
```

Les variables sont plus souvent utilisées pour enregistrer le résultat d'un calcul ou d'autres opérations utiles. Le résultat d'une addition, soustraction, multiplication ou division se fait comme suit

```
monAddition = 1 + 2.3
maSoustraction = 7.5 - 3
maMultiplication = 3.5 * 4
maDivision = 12 / 4
```

son affichage par la commande `print()` produira les résultats suivants

```
3.3
4.5
14.0
3.0
```

Si par la suite, on affecte une nouvelle valeur à une variable existante, la nouvelle valeur remplacera l'ancienne

```
maVariable = 'abc'
maVariable = 2
print(maVariable)
```

qui produira le résultat suivant

```
2
```

puisque la chaîne de caractères `abc` a été remplacée par le nombre 2 dans la variable `maVariable`.

En dehors des chiffres, des lettres et des chaînes de caractères, il existe aussi bien d'autres ***types*** de variables. Ceux qui nous intéresseront sont les ***objets***, les ***listes*** et les ***tuples***.

Dans la vie courante, un objet est une chose faite de matière et qui est façonné par l'Homme pour lui être utile. En Python, un objet sera représenté comme une variable et portera un nom comme une variable. Un objet peut avoir des propriétés : sa couleur, le matériau particulier utilisé, sa température, etc. Ce sont des propriétés qu'il ne partage pas avec d'autres objets en général mais pour des objets similaires, certaines de ces propriétés peuvent être les mêmes. En Python, on peut accéder aux propriétés d'un objet en plaçant un point après le nom de l'objet et le nom de la propriété comme ceci :

```
monObjet.materiau
monObjet.temperature
```

Un objet peut aussi avoir des fonctionnalités : une lampe peut s'allumer, un radiateur chauffe, un piano produit des sons, etc. Souvent, ces fonctionnalités consistent en des actions que réalise l'objet. En Python, ces actions propres à l'objet sont appelées ***méthodes*** et elles s'écrivent comme les propriétés mais avec des parenthèses à la fin. Elles sont similaires aux fonctions que nous verrons plus loin :

```
monPiano.jouer()
maLampe.allumer()
monRadiateur.chauffer()
```

Une liste est, comme son nom l'indique, une collection de données qui sont *ordonnées*, c'est à dire qui ont un ordre précis. Elle peut donc contenir des valeurs de tout type : des chiffres, des lettres et des chaînes de caractères, des objets, des tuples et même d'autres listes ! En Python, une liste est entourée par des crochets et ses éléments sont séparés par des virgules comme ceci :

```
maListe = [] # Liste vide
monAutreListe = [3, 5, 9, 2] # Liste uniquement de chiffres
mesLegumes = ["tomate", "patate", "chicon"] # Uniquement des chaînes de caractères
maListeFourreTout = [mesLegumes, "Hello", 5/3, ['hey', 3.14, 'oh!']]
```

On peut afficher un élément précis de la liste en précisant l'indice de l'élément voulu (compté à partir de 0!) entre crochets après le nom de la liste comme ceci:

```
print(monAutreListe)
print(maListeFourreTout[1]) # 2ème élément de la liste
print(mesLegumes[2]) # 3ème élément de la liste
```

Le résultat sera :

```
[3, 5, 9, 2]
Hello
chicon
```

Pour enlever un élément de la liste, on utilise la commande `del()` en indiquant la liste avec l'indice de l'élément à éliminer. Si tu veux enlever "Hello" de la liste fourre-tout, il faut donc écrire :

```
del(maListeFourreTout[1])
print(maListeFourreTout)
```

Pour ajouter un élément, il faut utiliser

Exercices

1. Change les indices de l'affichage de la liste et vois ce qu'il se produit. Que se passe-t-il lorsque l'indice est grand ?
2. Crée toutes les variables nécessaires et affiche le début du poème suivant de Victor Hugo :

```
Demain, dès l'aube, à l'heure où blanchit la campagne,
Je partirai. Vois-tu, je sais que tu m'attends.
J'irai par la forêt, j'irai par la montagne.
Je ne puis demeurer loin de toi plus longtemps.
```

Les commentaires

Dans les projets de codage, il est utile de pouvoir décrire ce qu'une ligne ou un bloc de code réalise. Cela permet de se souvenir rapidement d'utilité d'un morceau de code sans devoir le lire et le déchiffrer. Un autre avantage est que cela permet à un autre camarade de lire ton code et de le comprendre grâce aux commentaires.

Python considère que tout ce qui est écrit après un `#` est un commentaire

```
# Ceci est un commentaire en début de ligne
a = 1 # Ceci est un commentaire après une instruction
print(a) # Ceci est un autre commentaire après une instruction
```

donc à l'exécution ces commentaires seront simplement ignorés et son résultat sera

```
1
```

Dans la suite, on utilisera des commentaires aussi souvent que possible. C'est une bonne habitude à prendre quand on apprend à programmer.

Les fonctions

Les fonctions sont des commandes personnalisées que l'on construit soit même pour qu'il effectue une tâche particulière. La commande `print()` utilisé jusqu'à présent est une de ces commandes dont le seul objectif est d'afficher le contenu de la variable indiquée entre ses parenthèses.

De la même manière, on peut créer une fonction qui va réaliser un calcul particulier sur base d'une ou plusieurs informations nécessaires. Par exemple, on doit connaître le rayon d'un cercle pour déterminer son périmètre. La formule du périmètre du cercle est dans ce cas

$$\text{PerimetreCercle} = 2\pi \cdot \text{Rayon}$$

où $\pi \approx 3.14$. On pourrait écrire directement cette formule en Python comme ceci

```
pi = 3.14 # La constante mathématique
Rayon = 5 # Le rayon du cercle
PerimetreCercle = 2 * pi * Rayon # PérimetreCercle contient le résultat du calcul
```

mais comme tu le constates, il faut créer plusieurs variables. De plus, à chaque fois que l'on voudra calculer le rayon d'un cercle différent, il faudra changer la variable `Rayon` et ce n'est pas très pratique.

Pour créer cette fonction, il faut utiliser le mot-clé `def` suivi du nom de la fonction que l'on va créer, de parenthèses et finalement d'un deux-points `:` comme ceci

```
# Définition de ma fonction PerimetreCercle
def PerimetreCercle():
  # On écrit ici les étapes du calcul
  # Remarque les 2 espaces ajoutés ici!
  Perimetre = 2 * 3.14 * Rayon
  # On affiche le résultat du calcul avec un message
  print("Le périmètre vaut ")
  print(Perimetre)
```

Remarque que les instructions dans la fonction sont légèrement décalés de deux espaces et ceci est très important ! Ce décalage est appelé ***indentation*** et est primordial en Python car il permet de faire la différence entre les instructions de la fonction et ceux qui ne le sont pas. Si l'on tente d'exécuter le code comme tel, rien ne se produit. Pour utiliser la nouvelle fonction créée, il faut l'appeler comme on l'a fait avec `print()`, c'est à dire qu'il faut écrire le nom de la fonction suivit des parenthèses.

```
PerimetreCercle() # La fonction PerimetreCercle est appelée
```

Si l'on exécute ce code, il produira une erreur

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

qui indique que l'erreur est dûe à la variable `Rayon` qui n'est pas définie dans la fonction `PerimetreCercle()`. En effet, la fonction a besoin qu'on lui donne la valeur du rayon pour qu'il puisse terminer son calcul. Pour que ça soit le cas, il faut définir la fonction comme suit

```
def PerimetreCercle(Rayon):
  # On écrit ici les étapes du calcul
  # Remarque les 2 espaces ajoutés ici!
  Perimetre = 2 * 3.14 * Rayon
  # On affiche le résultat du calcul avec un message
  print("Le périmètre vaut ")
  print(Perimetre)
```

Comme tu le constates, il faut lui indiquer entre parenthèses qu'une valeur enregistré dans la variable nommé `Rayon` doit lui être communiqué lorsqu'on exécute la fonction. Maintenant, lorsqu'on voudra calculer le périmètre d'un cercle, on appellera la fonction comme suit

```
# Méthode 1: On précise la valeur de Rayon entre parenthèses
PerimetreCercle(5)
# Méthode 2:
# On crée une variable Rayon contenant la valeur
monRayon = 4
# Puis on appelle la fonction en indiquant cette variable entre parenthèses
PerimetreCercle(monRayon)
```

A l'exécution, on aura

```
Le périmètre vaut
31.400000000000002
Le périmètre vaut
25.12
```

Jusqu'à présent, le résultat du calcul de la fonction était affichée par la fonction elle-même avec `print()` mais si on veut récupérer le résultat pour l'utiliser plus tard au lieu de l'afficher tout de suite, comment faire? Dans une fonction, cela est rendu possible avec l'instruction `return` qui permet de renvoyer tout ce que l'on veut : valeur, variable, chaîne de caractères, etc. Ce résultat peut alors être enregistré dans une variable. Dans l'exemple précédent, on peut changer notre fonction `PerimetreCercle()` de la façon suivante

```
def PerimetreCercle(Rayon):
  # On écrit ici les étapes du calcul
  # Remarque les 2 espaces ajoutés ici!
  Perimetre = 2 * 3.14 * Rayon
  # On renvoie le résultat du calcul sans l'afficher
  return Perimetre
```

et à son appel, la fonction ne va plus afficher le résultat du calcul mais va le renvoyer. Il faudra alors enregistrer le résultat dans une variable comme ceci

```
# La variable monPerimetre contient la valeur 31.400000000000002
monPerimetre = PerimetreCercle(5)
print(monPerimetre)
```

A l'exécution, on a bien

```
31.400000000000002
```

Exercices

1. Crée une fonction qui affiche un triangle faite d'astérisques comme ceci :

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

2. Crée une fonction qui calcule le volume d'une sphère et le renvoie.

Le contrôle de flux

Jusqu'à présent, on a appris à créer quelques fonctions simples réalisant des opérations simples sur des variables. La fonction du calcul du périmètre ne fonctionne que pour un cercle et pour le périmètre d'un carré, il faudra créer une nouvelle fonction à nouveau. Comme ce serait pratique si on pouvait simplement dire à la fonction "Je veux un périmètre de cercle" ou "Je veux un périmètre d'un carré" et qu'il puisse adapter son calcul !

Heureusement, il existe une instruction Python qui permet de considérer des situations différentes en évaluant ***une condition***. Ce dernier est une opération dont le résultat ne peut être que ***Vrai*** (`True` en Python) ou ***Faux*** (`False` en Python) et il se réalise avec des opérateurs de comparaison dont une liste est présentée ci-dessous

Opérateur	Description
`A > B`	A plus grand que B
`A >= B`	A plus grand ou égal à B
`A < B`	A plus petit que B
`A <= B`	A plus petit ou égale à B
`A == B`	A égale/identique à B
`A != B`	A différent de B

Si... alors...

Les deux premières instructions sont `if` (***si*** en anglais) et `else` (***sinon*** en anglais) qui permet de gérer les situations du type "***si*** on me demande un périmètre de cercle alors j'utilise la formule $2\pi R$ ***sinon*** j'utilise la formule du périmètre d'un carré". En Python, on écrirait cela comme suit

```
if condition:
  instruction1
  instruction2
  instruction3
else:
  autre_instruction1
  autre_instruction2
```

Ainsi, si la condition est vraie alors les instructions suivantes seront exécutées. Si la condition est fausse alors d'autres instructions seront exécutées.

Dans notre cas, on voudrait créer une fonction `Perimetre()` qui soit capable de calculer le périmètre d'un cercle ou celui d'un carré simplement en lui indiquant "cercle" ou "carré" en argument avec la dimension du rayon ou du côté du carré. On pourrait le définir comme ceci

```
def Perimetre(type, dimension):
```

La première étape serait de vérifier que le type de forme est bien carré ou cercle. On utilise l'instruction `if` et la condition est de vérifier que la chaîne de caractère de la variable `type` corresponde bien à `"cercle"` avec l'opérateur de comparaison `==` puis on fait la même chose pour `"carre"` sinon on calcule le périmètre pour un carré. On le fait comme ceci

```
def Perimetre(type, dimension):
  if type == "cercle": # Si c'est un cercle
    resultat = 2 * 3.14 * dimension
  if type == "carre": # Si c'est un carré
    resultat = 4 * dimension
  else:
    print("Le type indiqué est incorrect!")
```

Il ne reste plus qu'à afficher le résultat avec un petit message

```
def Perimetre(type, dimension):
  if type == "cercle": # Si c'est un cercle
    resultat = 2 * 3.14 * dimension
    print("Périmètre du cercle: ")
    print(resultat)
  if type == "carre": # Si c'est un carré
    resultat = 4 * dimension
    print("Périmètre du carré: ")
    print(resultat)
  else:
    print("Le type indiqué est incorrect!")
```

Pour exécuter cette nouvelle fonction, il suffit de l'appeler en indiquant en argument ce que l'on veut. Par exemple, imaginons que l'on veuille calculer le périmètre d'un carré qu'on appelle la fonction comme ceci :

```
Perimetre("carré", 5)
```

A l'exécution, le message `Le type indiqué est incorrect!`. C'est que Python est très pointilleux sur les lettres qu'il compare. En effet, la condition dans `Perimetre` était `type == "carre"` et que le mot passé en argument était `"carré"` donc pour Python la simple présence de `é` au lieu de `e` rend le mot différent et la condition est évalué comme fausse (`False` dans Python). C'est pour cela que le message d'erreur est affiché. On aura le même résultat si on a le malheur d'appeler la fonction ainsi :

```
Perimetre("Carre", 5)
```

En effet, les majuscules comptent aussi comme une différence pour Python et la condition sera donc évaluée à `False`. La façon correcte d'appeler la fonction est de ce fait :

```
Perimetre("carre", 5)
```

Les boucles for

Dans certaines situations, il peut être intéressant de répéter plusieurs fois certaines opérations répétitives. C'est dans ce cadre que les boucles interviennent.

Les boucles `for` sont des boucles destinées à se répéter un nombre limité de fois. En Python, cela est possible en itérant sur une liste d'éléments. Voici un exemple pour illustrer le propos : tu es au parc pour un pic-nique et dans ta boite à tartines tu as trois tartines. Comme tu es très affamé, tu en prends un... puis un second... puis le dernier successivement et tu t'arrêtes car il ne reste plus rien dans ta boite à tartines. Bravo, tu viens d'exécuter une boucle ! Tu as *itéré* sur chaque tartine de ta "liste" (ici ta boite à tartines) pour effectuer un tâche répétitive, celui de manger la tartine choisie ! En Python, cela peut s'écrire de la façon suivante (en pseudo-code):

```
for tartine in boite_a_tartine:
  manger(tartine)
```

Ce code se lit à peu près comme *"pour chaque tartine dans la boite à tartine, mange la tartine»*. Comme suggéré, la boite à tartine sera en réalité une ***liste*** ou un ***tuple*** Python.

Imaginons que nous disposons de différentes variétés de tartines, ordonné comme ceci :

```
boite_a_tartine = ["thon mayo", "jambon beurre", "fromage"]
```

Alors pour afficher les tartines dans l'ordre, une boucle `for` sera utilisé comme ceci (en pseudo-code) :

```
for tartine in boite_a_tartine:
  print(tartine)
```

Cela produira le résultat suivant :

```
thon mayo
jambon beurre
fromage
```

Les boucles while

Tu cours pour un marathon de 20km et cela fait 30 min que tu cours. Comment tu sais quand tu dois t'arrêter de courir ? La question a l'air bête mais pour un ordinateur, il n'a aucun moyen de comprendre la situation si on ne lui décortique pas le problème jusqu'au moindre détail sous forme d'instructions claires. Pour une personne quelconque, il est évident qu'on a fini la course quand on

arrive à la ligne d'arrivée mais rien n'est évident pour un ordinateur, ***il faut tout lui expliciter***.

Si un robot prenait part à ce marathon, il faudrait que son programme prévoie qu'il s'arrête lorsqu'il arrive à la ligne d'arrivée donc tu codes son programme comme ceci (en pseudo-code):

```
courir()
if ligne_arrivee:
  arreter_de_courrir()
```

Mais voilà que le robot arrive à la ligne d'arrivée et ne s'arrête pourtant pas de courir! Comment ça se fait ? En réalité, le robot n'a fait qu'exécuter exactement ce que tu lui as demandé de faire. Il court et tout de suite, il vérifie s'il est à la ligne d'arrivée et évidemment il n'y est pas encore arrivé donc il ne s'arrête pas et c'est tout. Il ne vérifie pas par la suite s'il est arrivé à la ligne d'arrivée donc quand vient le moment de s'arrêter, il ne le fait pas puisque son code ne lui a pas demandé de vérifier à nouveau !

C'est dans ce cadre qu'interviennent les boucles `while` (*"tant que"* en français) : elles permettent à un processus de se dérouler ***tant qu***'une condition reste vraie. En Python, cela s'écrit comme ceci :

```
while condition:
  instruction1
  instruction2
  ...
```

Pour notre robot, le code adapté sera donc (en pseudo-code):

```
# Tant qu'on n'est pas arrivé à la ligne d'arrivée
while not ligne_arrivee:
  courrir() # ...on cours


# On est sorti de la boucle
arreter_de_courrir()
```

Un code simpliste qui fonctionne serait par exemple :

```
distance = 0 # Au départ 0 km sont parcourus


while distance < 20: # Tant que la distance < 20 km
```

```
  print("Je cours 1 km!")
  distance = distance + 1 # On a parcouru 1 km!

# On a fini de courir
print("J'ai fini! J'ai parcouru")
print(distance)
print("km")
```

Voici à présent un exemple plus mathématique : tu déposes 1000 euros sur ton compte épargne d'une banque qui promet un taux d'intérêt de 2% par mois. Cela veut dire que tu gagnes 2% de la somme disponible dans ton compte épargne chaque mois. Combien d'argent auras-tu dans ce compte après 24 mois (c'est à dire 2 ans) ? La condition de la boucle sera de vérifier combien de mois se sont écoulés et dans la boucle la somme disponible dans le compte épargne sera multiplié par 100%+2%=1.02. Donc le code est :

```
somme_epargne = 1000 # 1000 euros au départ
mois = 0 # On commence à compter les mois à partir de 0
while mois < 24:
  somme_epargne = somme_epargne * 1.02
  mois = mois + 1

print("La somme sur le compte épargne après 24 mois est")
print(somme_epargne)
```

Exercices

1. Crée une fonction qui affiche un triangle faite d'astérisques et correspondant à la hauteur indiqué en argument de la fonction:

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

2. Crée une fonction qui calcule le solde de ton compte épargne pour un nombre de mois et pour un taux d'intérêt indiqué en argument de la fonction.

Pour aller plus loin

Ce tutoriel n'était malheureusement qu'une introduction et ne remplace pas correctement un cours plus détaillé et complet de Python. Voici donc quelques références pour apprendre sur l'informatique et Python :

- ***OpenClassrooms.com*: Apprenez les bases du langage Python**
- ***Developpez.com*: Liste des meilleurs tutoriels Python en français**

Voici des sites qui proposent des cours complets sur Python mais aussi bien d'autres domaines à des niveau variant du débutant à expert:

- **EdX**
- **Coursera**
- **MIT Open Courseware**
- **Khan Academy**