Introduction à Python

Activité 01

Partie 1 : premiers pas pour la syntaxe en Python

Exercice 1 : opérations numériques élémentaires

1. Se connecter sur sa session, et ouvrir le logiciel Pyzo.
2. La fenêtre avec des `>>>` est **l'invite de commande** : c'est l'interface qui exécutera les calculs que vous souhaitez faire.

Les opérations élémentaires sont l'addition (`+`), la multiplication (`*`), la soustraction (`-`) et la puissance (`**`).

3. Calculer $5^{12} - 2^4 \times (3456 + 9 \times 2437) \times 601$.

Il y a deux divisions en Python, la division réelle (notée `/`) et la division entière (notée `//`).

4. Tester et comparer `1234/7` et `1234//7`. Quel est l'autre nom de la division entière ?
5. Calculer $\dfrac{7^9 - 2}{\dfrac{36 \times 385 \times 44\,188}{88} + 239 \times 4\,649}$.
6. Avec ces opérations élémentaires, calculer $\sqrt{101}$.

La dernière opération numérique élémentaire est `x % y`, qui permet d'obtenir ce que vaut `x` modulo `y`.

7. Calculer $777\,888\,999$ modulo 12345.
8. Est-ce que $1\,022\,545\,253$ est divisible par $14\,821$? Et est-ce que $897\,036\,709$ est divisible par $11\,587$?

Exercice 2 : appeler des fonctions et des bibliothèques

En informatique, tous les calculs ne peuvent être faits (raisonnablement) uniquement à l'aide d'opérateurs : il faut aussi appeler des **fonctions**. Heureusement, la syntaxe est très proche de celle des mathématiques : une fonction `f` est appelée sur l'entrée `x` en écrivant `f(x)`.

1. La fonction `abs` renvoie la valeur absolue d'un nombre. Calculer $|6\,765 \times 17\,711 - 10\,946^2|$.

Toutes les opérations mathématiques classiques ne sont pas implémentées dans les commandes de base de Python : il faut appeler des fonctions externes, stockées dans des **bibliothèques**.

2. Tester la commande `cos(100)`. Lire le message d'erreur : que vous dit-il ?
3. Maintenant, exécuter la commande « `from math import cos` ». Puis tester la commande `cos(100)`.

Ici, `math` est une bibliothèque, c'est-à-dire un fichier annexe que Python ne charge pas par défaut, mais chargera si on lui demande. Plus précisément, vous avez ici demandé à Python de chercher une fonction précise, `cos`, dans le fichier `math`.

4. Fermer la fenêtre Pyzo, et tester de nouveau `cos(100)`, sans écrire l'importation. Qu'en conclure ?

Pour avoir accès à toutes les fonctions contenues dans `math`, utiliser la commande « `from math import *` ».

5. Calculer $\ln\left(\dfrac{\exp(\tan(\mathrm{Arccos}(0.3)))}{\sqrt{\cos(81)}}\right)$ à l'aide des fonctions `log`, `exp`, `tan`, `sqrt`, `cos` et `acos`.

Une fonction peut avoir plusieurs paramètres en entrée : par exemple, `pow(a,b)` permet de calculer a^b.

6. Tester `pow(1953125,1/9)`. Que venez-vous de calculer ? Y a-t-il une autre syntaxe pour faire ce calcul ?

Partie 2 : écrire ses propres fonctions

Exercice 3

Maintenant, vous savez utiliser Python comme une calculatrice très encombrante : l'étape suivante consiste à programmer *pour de vrai !*

1. La grande fenêtre de gauche est un fichier Python vierge qui n'est pas encore sauvegardé. Sauvegarder le fichier sous le nom `TP1` dans votre dossier réseau personnel.

Jusque-là, nous avons appelé des fonctions préexistantes : il s'agit maintenant d'en créer une de vos propres mains.

Pour définir une fonction, il faut utiliser le mot clé `def` : « `def` *<nom_fonction>*`(`*<paramètre>*`)` ». Pour que la fonction renvoie un résultat, il faut utiliser le mot-clé `return`. Attention à l'espacement : tout le code à l'intérieur de la fonction doit être tabulé (touche à gauche du « a » sur le clavier).

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

2. Recopier la fonction précédente dans le fichier `TP1` ; appuyer sur « Ctrl + E » (il devrait y avoir une ligne de texte qui s'affiche dans l'invite de commandes) ; tester la commande `double(255)`.
3. Écrire une fonction `carre(n)` qui prend en entrée un nombre et qui renvoie le nombre considéré au carré. On remarque que la fonction `double` et la fonction `carre` utilisent toutes les deux un paramètre `n` : chaque fonction « existe dans son monde », et il n'y a pas d'incidence entre les deux fonctions, qui fonctionnent indépendamment même si certains paramètres partagent un même nom.
4. Une fonction peut avoir plusieurs paramètres : écrire une fonction `somme_div_eucli(n,p)` qui renvoie la somme du quotient et du reste de la division euclidenne de `n` par `p`.

Pour utiliser des fonctions provenant de bibliothèques, le mieux est d'importer ce que vous souhaitez à l'intérieur du fichier Python. Techniquement, si vous avez déjà fait vos importations dans l'invite de commande, elles seront accessibles : mais si vous refermez le fichier, vous devrez recommencer vos importations.

```
from math import *

def trigo(d):
    return cos(d)**2 - sin(d)**2
```

5. Écrire une fonction `distance(x1,y1,x2,y2)` qui renvoie la distance entre les points de coordonnées (x_1, y_1) et (x_2, y_2) dans le plan euclidien en utilisant la fonction `sqrt` de la bibliothèque `math`.
6. Écrire une fonction `discriminant(a,b,c)` qui renvoie le discriminant de l'équation du second degré $ax^2 + bx + c = 0$.

Les fonctions que vous écrivez peuvent non seulement appeler d'autres fonctions préexistantes, mais aussi **vos propres fonctions**. Ici, la fonction `quadrupleplusun` demande à la fonction `quadruple` de lui renvoyer le résultat avec l'argument `2n`, récupère ce résultat et lui ajoute 1 avant de renvoyer le tout.

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

def quadrupleplusun(n):
    return double(2*n)+1
```

7. Écrire une fonction `solution(a,b,c)` qui renvoie une solution réelle à l'équation $ax^2 + bx + c = 0$, en appelant la fonction `discriminant` écrite précédemment. Tester `solution` avec l'argument `a = 2, b = -3, c = 1`. Tester `solution avec l'argument a = 1, b = 0, c = 1`. Que se passe-t-il ? Lire attentivement le message d'erreur.

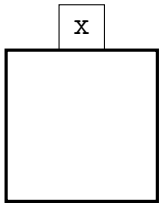
Partie 3 : variables

Exercice 4

En programmation, il est nécessaire d'utiliser des **variables**. Une variable est une case mémoire joker, dans laquelle vous pouvez stocker des données, même temporairement.
La principale opération pour modifier une variable s'appelle l'**affectation**. En Python, elle s'écrit avec le symbole `=` : attention, ce n'est pas le symbole d'égalité mathématique. Il vaut mieux l'interpréter comme « désormais, `x` vaut 2 ».
En pseudo-code, on écrit « `x` ← 2 » : l'interprétation est « on stocke la valeur 2 dans la variable `x` ».

1. Recopier ces lignes de code dans le fichier Python, et exécuter le fichier.
2. Exécuter le programme avec un « schéma boîte » : le carré ci-contre représente la variable `x`, dans laquelle vous mettrez les valeurs successives prises par `x`.

```
x = 2
print(x)
x = 4
print(x)
x = x + 3
print(x)
```



Exercice 5

Exécuter **à la main** (c'est-à-dire vous-même, sur une feuille avec un crayon) les programmes suivants à l'aide de schémas boîte, et après seulement les tester sur machine.

```
x = 9
y = 5
x = x + y
y = x + y
```

```
a = 17
b = 5
ab = a%b
ba = ab + a//b
```

```
n = 3
n = n**n+n
n = (n+2)**(1/5)
```

Partie 4 : ma première boucle

Exercice 6

Enfin, pour pouvoir faire des programmes qui servent à quelque chose, nous allons avoir besoin de faire des **boucles**, c'est-à-dire des actions répétitives. Nous verrons, la semaine prochaine, comment ces boucles fonctionnent plus précisément : pour le moment, nous allons nous concentrer sur un modèle très simple.

1. Tester le code ci-contre.
2. Adapter le code pour que soient affichés les entiers de 0 à 10 compris.
3. Adapter le code pour que soit affiché « `"youpi"` » 10 fois d'affilée.

```
for i in range(10):
    print(i)
```

Exercice 7

Une première technique de programmation essentielle en informatique est l'utilisation d'un **accumulateur**.

1. Exécuter à la main le programme suivant, puis le tester sur machine.
2. Écrire une fonction dont le nom est `somme` qui prend en paramètre `n` et **retourne** la sommme des entiers de 0 à `n` inclus.

```
s = 0
for i in range(5):
    s = s + i
print(s)
```

Exercice 8

En Python, vous avez pu remarquer que l'indentation (la façon dont le code est espacé à gauche) influence son interprétation par la machine : l'indentation délimite des **blocs de commande**. Ces blocs de commande influencent notamment les boucles et les fonctions.
Une boucle « pour » répète exactement le bloc de commande associé; appeler une fonction exécute exactement le bloc de commande associé, et les variables définies dans le bloc d'une fonction ne sont pas définies en dehors.

```
def produit(tab):
    compteur = 1
    for element in tab :
        compteur *= element
    return compteur
```

1. Dessiner les blocs de commande associé aux programmes suivants.

```
x = 2
for i in range(5):
    x = x + i
    for j in range(6):
        x = x * (1+j)
    print(x)
```

```
x = 2
for i in range(5):
    x = x + i
    for j in range(6):
        x = x * (1+j)
print(x)
```

```
def calcul(n):
    c = 1
    for i in range(n):
        c = c * (1+i)
    return c

calcul(5)
print(c)
```

2. Sans exécuter sur machine, lequel des deux premiers programmes fait plusieurs affichages, et lequel un seul ?
3. Tester le troisième programme. Que se passe-t-il ? Lire attentivement le message d'erreur, et l'analyser à l'aide des blocs de commande.

Exercice 9

Lors de la manipulation d'une fonction, le `return` et le `print` peuvent sembler identiques, mais ont des différences profondes de fonctionnement.

1. Tester le premier programme. En particulier, dans l'invite de commandes, appeler la fonction `nb()`.
2. Tester le second. En particulier, dans l'invite de commandes, appeler la fonction `nb2()`.

```
def nb():
    return 3

def double(n):
    return 2*n

print(double(nb()))
```

```
def nb2():
    print(3)

def double(n):
    return 2*n

print(double(nb2()))
```

Exercice 10

Une autre propriété importante du `return` est qu'il interrompt la fonction en cours.

1. Tester le premier programme, en appelant `fonction_test()` dans l'invite de commandes : que constatez-vous ?

```
def fonction_test():
    print("Coucou !")
    return 5
    print("Me voilà !")
```

Partie 5 : et si on danse ?

Exercice 11

Une autre structure importante en programmation est la **conditionnelle** : si une condition est remplie, on exécute un morceau de code ; sinon, on ne l'exécute pas, ou on en exécute un autre.

```
def est_pair(n):
    if n%2==0 :
        return True
    else :
        return False
```

1. Tester le programme ci-contre.
2. Écrire une fonction `admet_solution_reelle(a,b,c)` qui renvoie `True` si l'équation $ax^2 + bx + c = 0$ admet au moins une solution réelle, et `False` sinon.

Exercice 12

La condition précédente « `n%2==0` » est appelée un **booléen** : soit c'est vrai, soit c'est faux. En Python, on peut combiner des booléens avec `and`, `or` et `not` : la condition `n%2==0 and n%3==0` est remplie si `n` est divisible par 2 **et** par 3.
Une année `n` est bissextile si elle vérifie l'une des conditions suivantes :

— elle est divisible par 4, mais pas par 100 ;

— ou alors, elle est divisble par 400.

Par exemple, 1600 est bissextile, mais 1800 ne l'est pas. Écrire une fonction `bissextile(n)` qui renvoie `True` si `n` est une année bissextile, et `False` sinon.

Exercice 13

1. Écrire une fonction `maximum2(a,b)` qui renvoie le maximum entre `a` et `b`.
2. En appelant la fonction précédente, écrire une fonction `maximum3(a,b,c)` qui renvoie le maximum entre `a`, `b` et `c`.

Exercice 14

Écrire une fonction `somme_multiple3ou5(n)` qui prend en entrée un entier `n` et renvoie la somme des multiples de 3 ou de 5 plus petits que `n`. Par exemple, pour `n` = 10, les multiples de 3 ou de 5 plus petits que 10 sont 0, 3, 5, 6, 9 et 10 : leur somme vaut 33.

Exercice 15

Écrire une fonction `meme_signe(x,y)` qui renvoie `True` si `x` et `y` sont de mêmes signes. Modifier votre code pour **ne pas utiliser de « si »**.

Pour la semaine prochaine :

— Écrire une fonction `fact` qui prend en entrée `n` et retourne la factorielle de `n`, c'est-à-dire $1 \times 2 \times 3 \times \cdots \times n$.

— Écrire une fonction `somme_restes` qui prend en entrée deux entiers `n` et `m` et renvoie la somme des entiers modulo `m` compris entre 0 et `n` inclus. Par exemple, pour `n` = 5 et `m` = 3, la fonction renverra $(0 \bmod 3) + (1 \bmod 3) + (2 \bmod 3) + (3 \bmod 3) + (4 \bmod 3) + (5 \bmod 3) = 6$.

— En appelant la fonction `fact` écrite avant, écrire une fonction `somme_fact` qui prend en entrée un entier `n` et renvoie la somme des factorielles de 1 à `n` inclus. Par exemple, pour `n` = 4, la fonction renverra $1!+2!+3!+4! = 33$.

— Écrire une fonction `triangle_rect(a,b,c)` qui renvoie `True` s'il existe un triangle rectangle de côtés `a`, `b` et `c`.

— Écrire une fonction `triangle(a,b,c)` qui renvoie `True` s'il existe un triangle de côtés `a`, `b` et `c` (par exemple, il n'existe pas de triangle de côtés 1, 2 et 35).