Introduction

Intervenants en CI (*bureau 225* du bâtiment de l'UFR) :

- Emmanuel Franck (`emmanuel.franck@inria.fr`)
- Victor Michel-Dansac (`victor.michel-dansac@math.unistra.fr`)

Intervenants en TP :

- Léo Bois (`l.bois@math.unistra.fr`)
- Victor Michel-Dansac (`victor.michel-dansac@math.unistra.fr`)
- Guillaume Steimer (`g.steimer@unistra.fr`)

En L1, ont été introduites les notions de base de programmation. L'approche introduite est appelée l'approche impérative. Les programmes se décomposent en séries d'instructions de base (comme les boucles ou les tests).
Plus précisement, vous avez étudié une approche procédurale qui va décomposer ce programme en plusieurs sous-programmes ou **fonctions** (ce qui est très utile pour réutiliser du code). Le programme est ensuite constitué d'appels sucessifs à ces fonctions.

Il existe d'autres approches (ou paradigmes) de programmation, notamment la programmation orientée objet.

Ce cours : *Introduction à la programmation objet*.

Langages de programmation

- **Langages centraux en programmation objet** : C++ (L3 Mathématiques Appliquées) très utilisé pour le calcul (simulation, apprentissage profond, etc.), Java très utilisé pour le développement logiciel.
- **Python** : langage de haut niveau, non typé, non compilé (contrairemenent au Java et au C++) capable de faire de la programmation objet, et très utilisé en sciences (mathématiques, statistique, biologie, IA et apprentissage, ...).

Ce cours : *Programmation orientée objet à travers le Python*.

1 : Rappels de Python

1.1 : Premier code, variable et affectation

1er programme : Affichage d'une date

In [1]:
```
x = 15
print("Nous sommes le ", x, " septembre 2021.")
# print est une fonction de base de Python
```

```
Nous sommes le  15  septembre 2021.
```

Dans ce programme :

- on définit une **variable** appelée x,
- on lui **affecte** la valeur 15 avec l'**opérateur** =,
- on affiche un message contenant la valeur de x,
- avec # on peut écrire des commentaires.

2ème programme : Calcul numérique

In [2]:

```
x = 12.0
y = (1.5 * x) + 1.3
z = y**2.0  # l'opérateur ** représente la puissance
print("y vaut ", y, " ; z vaut ", z)
```

```
y vaut  19.3  ; z vaut  372.49
```

En mathématiques, le Python est utilisé pour faire du **calcul numérique** sur les nombres.

Mais aussi:

- de faire du calcul numérique sur des vecteurs/matrices (bibliothèque `numpy`),
- d'afficher des résultats (bibliothèque `matplotlib`),
- de résoudre d'autres problèmes mathématiques (bibliothèques `scipy`, `tensorflow`, etc.).

1.2 : Types de base

On va maintenant rappeler les types de données de base du Python. On rappelle que le Python est non typé. On n'a pas besoin d'indiquer, au moment de **déclarer une variable**, le type de cette variable : le programme va le **deviner**.

Les types simples les plus courants sont :

- `int` (nombre entier relatif, *integer* en anglais),
- `float` (représente un nombre réel),
- `str` (chaîne de caractères (mot), *string* en anglais),
- `bool` (booléen, pouvant être vrai ou faux).

In [3]:

```
x = 1
print(" type de x: ", type(x), "   ; valeur de x: ", x)

y = 3.14
print(" type de y: ", type(y), " ; valeur de y: ", y)

z = "coucou"
print(" type de z: ", type(z), "   ; valeur de z: ", z)

a = True
print(" type de a: ", type(a), "  ; valeur de a: ", a)
```

```
 type de x:  <class 'int'>    ; valeur de x:  1
 type de y:  <class 'float'>  ; valeur de y:  3.14
 type de z:  <class 'str'>    ; valeur de z:  coucou
 type de a:  <class 'bool'>   ; valeur de a:  True
```

Dans certains langages, on doit préciser le **type** d'une variable à sa première utilisation d'une **variable** (si c'est un `int` ou un `float` par exemple). On parle alors de déclaration. Une fois déclarée, une variable ne peut pas facilement changer de type.

En **`Python`**, ce changement de type est possible car on n'est pas obligé de déclarer/typer une variable. Le langage interprète. Ici, $x = 1$ est interprété comme un **integer**, $y = 3.14$ comme un **float** et le produit entre les deux comme un **float**.

1.3 : Types complexes

On va maintenant introduire des types plus complexes. Il s'agit de tableaux permettant de traiter d'un seul coup un ensemble de données.

Les types complexes présents dans `Python` sont:

- les listes,
- les tuples,
- les dictionnaires.

1.3.1 : Listes

Les listes : *Une* liste *est une collection* ***ordonnée*** *et* ***modifiable*** *d'éléments, éventuellement hétérogènes*.

Il s'agit d'un type **complexe** fourni par Python.

Exemple:

In [4]:

```
couleurs = ["trèfle", "carreau", "coeur", "pique"]
print("couleurs = ", couleurs)
couleurs[0] = 1.2
print("couleurs = ", couleurs)
```

```
couleurs =  ['trèfle', 'carreau', 'coeur', 'pique']
couleurs =  [1.2, 'carreau', 'coeur', 'pique']
```

La liste `couleurs` contient au départ des **chaînes de caractères**. Puisqu'une liste peut être hétérogène, on peut ensuite remplacer le 1er élement par une valeur d'un autre type (ici un `float`).

Important: *Les éléments d'une liste sont indexés en commençant à zéro*.

Un certain nombre d'opérations sur ces listes sont possibles.

In [4]:
```
couleurs = ["trèfle", "carreau", "carreau", "pique"]
print("couleurs = ", couleurs)
couleurs.append(1.7)  # ajoute 1.7 à la fin de la liste
print("couleurs = ", couleurs)
couleurs.reverse()  # inverse l'odre de parcours de la liste
print("couleurs = ", couleurs)
couleurs.remove("carreau")  # enlève la chaîne de caractères "carreau" de la liste
print("couleurs = ", couleurs)
couleurs.append(1.7)  # ajoute 1.7 à la fin de la liste
print("couleurs = ", couleurs)
c = couleurs.count(1.7)  # compte le nombre de 1.7 dans la liste
print("couleurs.count(1.7) = ", c)
```

```
couleurs =  ['trèfle', 'carreau', 'carreau', 'pique']
couleurs =  ['trèfle', 'carreau', 'carreau', 'pique', 1.7]
couleurs =  [1.7, 'pique', 'carreau', 'carreau', 'trèfle']
couleurs =  [1.7, 'pique', 'carreau', 'trèfle']
couleurs =  [1.7, 'pique', 'carreau', 'trèfle', 1.7]
couleurs.count(1.7) =  2
```

- `append` : ajout un élément en fin de liste,
- `reverse` : inverse la liste,
- `remove` : enlève un certain élément,
- `count` : compte le nombre de fois qu'est présent un élément.

Il existe d'autres fonctions permettant d'agir sur une **liste**. Les traitements de la forme `variable.f()` sont appelés des **méthodes** (on y reviendra dans un chapitre suivant).

On peut travailler sur des **tranches** d'une liste.

In [2]:
```
couleurs = ["trèfle", "carreau", "coeur", "pique"]
print("couleurs = ", couleurs)

print("couleurs[0:2] = ", couleurs[0:2])

couleurs[0:2] = [1.2, "coucou", 3.7]
print("couleurs = ", couleurs)
```

```
couleurs =  ['trèfle', 'carreau', 'coeur', 'pique']
couleurs[0:2] =  ['trèfle', 'carreau']
couleurs =  [1.2, 'coucou', 3.7, 'coeur', 'pique']
```

On va maintenant parler d'une notion pas évidente : le passage ou la copie par valeur ou par réference.

Exemples sur les nombres:

In [7]:
```
a = 2
b = a
print("a (avant modification de a) = ", a)
print("b (avant modification de a) = ", b)
a = 3
print("\na (après modification de a) = ", a)
print("b (après modification de a) = ", b)
```

```
a (avant modification de a) =  2
b (avant modification de a) =  2

a (après modification de a) =  3
b (après modification de a) =  2
```

Dans cet exemple, `a = 2` crée le nombre entier 2 et la variable `a` pointant sur 2.

Ensuite, `b = a` crée une variable b qui pointe sur la valeur pointée actuellement par a, c'est-à-dire 2. La commande `a = 3` crée le nombre entier 3 et fait pointer la variable `a` dessus.

On peut alors vérifier que la variable `b` est inchangée.

Comment marche l'affection `=` ?

Son rôle est d'affecter une valeur à une variable. Donc remplir une variable (le contenant) avec une donnée (le contenu).

Dans la ligne 1 ci-dessus, l'affectation réalise plusieurs opérations :

- création en mémoire d’un objet du type approprié (membre de droite) ;
- stockage de la donnée dans l’objet créé ;
- création d’un nom de variable (membre de gauche) ;
- association de ce nom de variable avec l’objet contenant la valeur.

Exemples sur les listes:

In [8]:

```
print("Exemple 1 :\n")
fable = ["Je", "plie", "mais", "ne", "rompt", "point"]
phrase = fable

print("fable avant modification de phrase = \n", fable)
phrase = ["Je", "plie", "mais", "ne", "casse", "point"]
print("phrase = ", phrase)
print("fable après modification de phrase = \n", fable)
```

```
Exemple 1 :

fable avant modification de phrase =
 ['Je', 'plie', 'mais', 'ne', 'rompt', 'point']
phrase =  ['Je', 'plie', 'mais', 'ne', 'casse', 'point']
fable après modification de phrase =
 ['Je', 'plie', 'mais', 'ne', 'rompt', 'point']
```

Exemples sur les listes:

In [11]:

```
print("Exemple 2 :\n")
fable = ["Je", "plie", "mais", "ne", "rompt", "point"]
phrase = fable

print("fable avant modification de phrase = \n", fable)
phrase[4] = "casse"
print("phrase = ", phrase)
print("fable après modification de phrase = \n", fable)
```

```
Exemple 2 :

fable avant modification de phrase =
 ['Je', 'plie', 'mais', 'ne', 'rompt', 'point']
phrase =  ['Je', 'plie', 'mais', 'ne', 'casse', 'point']
fable après modification de phrase =
 ['Je', 'plie', 'mais', 'ne', 'casse', 'point']
```

Comment expliquer le comportement ci-dessus ? Dans un cas les deux variables subissent les modifications, et pas dans l'autre.

Dans le cas 1, c'est comme avant. On crée un objet `["Je", "plie", "mais", "ne", "casse", "point"]` et on va associer **`phrase`** à ce nouvel objet `["Je", "plie", "mais", "ne", "casse", "point"]`.

Dans le cas 2, *`phrase[4] = "casse"`* ne modifie qu'une partie de l'objet. Dans les deux cas : **`phrase`** et **`fable`** sont toujours associés à la même liste.

Ici, on ne copie pas le contenu de **`fable`** dans **`phrase`** mais **l'emplacement en mémoire de la liste**. On parle de copie par réference.

Si on veut copier le contenu de la liste sans que l'adresse soit copiée, il faut utiliser la méthode `copy`. On parle alors de copie par valeur ou de copie profonde.

In [10]:
```
fable = ["Je", "plie", "mais", "ne", "rompt", "point"]
phrase = fable.copy()

print("fable avant modification de phrase = \n", fable)
phrase[4] = "casse"
print("phrase = ", phrase)
print("fable après modification de phrase = \n", fable)
```

```
fable avant modification de phrase =
 ['Je', 'plie', 'mais', 'ne', 'rompt', 'point']
phrase =  ['Je', 'plie', 'mais', 'ne', 'casse', 'point']
fable après modification de phrase =
 ['Je', 'plie', 'mais', 'ne', 'rompt', 'point']
```

1.3.2 : Tuples

Les tuples: *Un* tuple *est une collection* ***ordonnée*** *et* ***non modifiable*** *d'éléments, éventuellement hétérogènes.*

Essayer de modifier un élément d'un tuple de façon analogue à la modification d'une liste résulte en une erreur.

In [11]:
```
tuple_vide = ()
couleurs2 = ("trèfle", "carreau", 1.2, "pique")
print(">>>", couleurs2)
couleurs2[2] = 1.7
```

```
>>> ('trèfle', 'carreau', 1.2, 'pique')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-c1cc5e1c6824> in <module>
      2 couleurs2 = ("trèfle", "carreau", 1.2, "pique")
      3 print(">>>", couleurs2)
----> 4 couleurs2[2] = 1.7

TypeError: 'tuple' object does not support item assignment
```

1.3.3 : Dictionnaires

Les listes ou les tuples sont des types complexes ou les données sont **ordonnées** et accessibles à partir d'un numéro : l'**indice de l'élément**. On parle de conteneurs de données séquentiels.

Les dictionnaires marchent différement. Les données ne sont pas ordonnées, et on y accède à l'aide d'une clé qui peut être un nombre, une chaîne de caractères, etc. On parle alors de conteneurs de données associatifs.

In [26]:

```
fonctions_dérivées = {"sinus": "cosinus", "exp": "exp"}
print(" dérivée de sinus :", fonctions_dérivées["sinus"])
print(" dérivée de exp :", fonctions_dérivées["exp"])
```

```
 dérivée de sinus : cosinus
 dérivée de exp : exp
```

Un dictionnaire en Python va permettre de rassembler des éléments, mais ceux-ci seront identifiés par une **clé** et pas par un indice comme dans une liste ordonnée.

On accède à un élément par sa **clé**. Ici, en donnant le nom de la fonction (clé), on obtient le non de la dérivée.

In [27]:
```
print(" dictionnaire : ", fonctions_dérivées)
fonctions_dérivées["log"] = "1/x"  # ajout d'un élement
print(" dictionnaire : ", fonctions_dérivées)
fonctions_dérivées["2x"] = 2  # ajout d'un élement
print(" dictionnaire : ", fonctions_dérivées)
```

```
 dictionnaire :  {'sinus': 'cosinus', 'exp': 'exp'}
 dictionnaire :  {'sinus': 'cosinus', 'exp': 'exp', 'log': '1/x'}
 dictionnaire :  {'sinus': 'cosinus', 'exp': 'exp', 'log': '1/x', '2x': 2}
```

Ceci montre comment ajouter des élements.

On peut aussi associer des objets de différents types: ici, à la clé `2x`, on a associé un entier et non une chaîne de caractères.

Type possible pour les "clés": entier, chaîne de caractères, tuple. On ne peut pas utiliser des `float` ou des listes.

En effet, on a besoin de pouvoir tester **l'égalité exacte** entre deux clés, ce qui n'est pas possible entre deux `float` à cause de la précision machine.

Remarque : Illustration de la **précision machine**

In [5]:
```
x = 1e-15
y = ((1 + x) - 1) / x
print(y)
```

```
1.1102230246251565
```

Ici, y devrait toujours valoir 1. Mais quand x devient petit, y se met à fluctuer autour de 1, puis tombe brutalement à 0.

Ce petit calcul est une illustration d'un concept clé en mathématiques numériques : la **précision machine**.

Parcours de dictionnaire :

In [28]:
```
print("Affichage des éléments contenus dans le dictionnaire :")
for i in fonctions_dérivées.items():
    print(i)

print("\nAffichage des clés et valeurs contenus dans le dictionnaire :")
for keys, items in fonctions_dérivées.items():
    print("clé: ", keys, ", valeur: ", items)
```

```
Affichage des éléments contenus dans le dictionnaire :
('sinus', 'cosinus')
('exp', 'exp')
('log', '1/x')
('2x', 2)

Affichage des clés et valeurs contenus dans le dictionnaire :
clé:  sinus , valeur:  cosinus
clé:  exp , valeur:  exp
clé:  log , valeur:  1/x
clé:  2x , valeur:  2
```

Recherche d'une clé dans un dictionnaire :

In [31]:
```
print("Recherche de l'existence d'une clé :")
if "log" in fonctions_dérivées:  # test si une clé est présente
    print("La clé 'log' existe")
```

```
Recherche de l'existence d'une clé :
La clé 'log' existe
```

1.4 : Tests

Pour faire des programmes plus complexes, on a besoin de **tests**.

Ceux-ci sont réalisés à partir de **booléens**, qui peuvent prendre les valeurs `True` (vrai) ou `False` (faux) (attention aux majuscules !). On peut ensuite tester la valeur d'un booléen avec la structure `if`, `elif`, `else`.

In [33]:

```
a = True

if a:
    print("a est vrai")
else:
    print("a est faux")

b = False

if b:
    print("b est vrai")
else:
    print("b est faux")
```

```
a est vrai
b est faux
```

La structure `if`, `elif`, `else` permet de faire des comparaisons de valeurs de variables.

Les opérateurs utiles pour de tels tests sont :

- `a == b` pour un test d'égalité (`a == b` renvoie `True` si `a = b`, `False` sinon)
- `a != b` pour un test de différence (`a != b` renvoie `False` si `a = b`, `True` sinon)
- `a < b` pour un test d'infériorité stricte, `a <= b` pour un test d'infériorité large
- `a > b` pour un test de supériorité stricte, `a >= b` pour un test de supériorité large

In [18]:

```
a = 2
b = 3

print("a = ", a, " ; b = ", b)
if a >= b:
    print("a est supérieur ou égal à b")
else:
    print("a est strictement inférieur à b\n")

a = 2
b = 2

print("a = ", a, " ; b = ", b)
if a < b:
    print("a est strictement inférieur à b")
elif a == b:
    print("a est égal à b")
else:
    print("a est strictement supérieur à b")
```

```
a =  2  ; b =  3
a est strictement inférieur à b

a =  2  ; b =  2
a est égal à b
```

On peut enfin utiliser les opérateurs `not`, `and` et `or` pour représenter, respectivement, le "non" logique, le "et" logique, et le "ou" logique.

In [19]:

```
a = 2
b = 3
c = 1
print("a = ", a, " ; b = ", b, " ; c = ", c)

if not (c > b):
    print("c est inférieur ou égal à b")
else:
    print("c strictement supérieur à b")

if (a < b) and (b < c):
    print("c est strictement supérieur à b ET b est strictement supérieur à a")

if (a < b) or (b < c):
    print("c est strictement supérieur à b OU b est strictement supérieur à a")
```

```
a =  2  ; b =  3  ; c =  1
c est inférieur ou égal à b
c est strictement supérieur à b OU b est strictement supérieur à a
```

1.5 : Boucles

Les **boucles** seront aussi essentielles pour écrire des programmes complexes. Elles permettent de répéter un jeu d'instructions sans avoir à copier/coller le code.

Il en existe deux types : les boucles `for` et les boucles `while`.

1.5.1 Boucles `for`

Les boucles `for` permettent d'itérer sur les éléments d'un tableau (liste, tuple, etc.).

Notamment, la fonction `range` est très utile pour faire une boucle sur des valeurs numériques, de `0` jusqu'à un certain nombre (ici, `3`).

In [20]:
```
print("Démonstration de range(4) : ")
for i in range(4):
    print(i)
```

```
Démonstration de range(4) :
0
1
2
3
```

Attention, quand on écrit `for i in range(4)`, `i` prend ses valeurs de 0 à 3 inclus, et ne va pas jusqu'à 4 !

De la même façon, on peut aussi spécifier là où `i` doit commencer (`début`) et le pas de l'itération (`pas`).

In [6]:
```
début = 3
fin = 13
pas = 4
# on va itérer de 3 à 13 (exclus) avec un pas de 4
print("Démonstration de range(", début, ",", fin, ",", pas, ")")
for i in range(début, fin, pas):
    print(i)
```

```
Démonstration de range( 3 , 13 , 4 )
3
7
11
```

Il est aussi possible d'itérer directement sur les éléments d'une liste.

In [22]:

```
fable = ["Je", "plie", "mais", "ne", "rompt", "point"]
print("La liste fable est : ", fable)

print("Boucle sur les éléments de la liste fable :")

for mot in fable:
    print(mot)
```

```
La liste fable est :  ['Je', 'plie', 'mais', 'ne', 'rompt', 'point']
Boucle sur les éléments de la liste fable :
Je
plie
mais
ne
rompt
point
```

La fonction `enumerate` permet d'un seul coup d'itérer sur les éléments de la liste et de récupérer leur indice.

In [23]:

```
fable = ["Je", "plie", "mais", "ne", "rompt", "point"]
print("La liste fable est : ", fable)

print("Les éléments de la liste fable et leurs indices sont :")

for indice, mot in enumerate(fable):
    print("élément numéro", indice, " : ", mot)
```

```
La liste fable est :  ['Je', 'plie', 'mais', 'ne', 'rompt', 'point']
Les éléments de la liste fable et leurs indices sont :
élément numéro 0  :  Je
élément numéro 1  :  plie
élément numéro 2  :  mais
élément numéro 3  :  ne
élément numéro 4  :  rompt
élément numéro 5  :  point
```

1.5.2 : Boucles `while`

Ces boucles permettent d'effectuer un jeu d'instructions tant qu'une certaine condition est vérifiée.

La boucle ci-dessous est équivalente à `for i in range(4)` :

In [39]:
```
i = 0
while i < 4:
    print(i)
    i += 1  # signifie i = i + 1
```

```
0
1
2
3
```

Attention aux points suivants :

- la boucle va tourner tant que la condition est vraie (ici, tant que `i` est inférieur à `4`) ;
- il donc faut penser à **mettre à jour** la condition à l'intérieur de la boucle (ici, le `i += 1` est nécessaire : s'il n'est pas présent, la condition `i < 4` sera toujours vraie, et la boucle ne s'arrêtera jamais : on parle alors de boucle infinie) ;
- il faut toujours **initialiser** la condition d'une boucle `while` (ici, en initialisant `i` à `0`).

Un exemple plus complexe en mathématiques : convergence de la suite $u_{n+1} = \sqrt{u_n}$ avec $u_0 = 3$.

In [59]:

```
import numpy as np  # bibliothèque contenant certaines fonctions mathématiques
u_n = 3  # initialisation de u_n à 3
k = 0  # initialisation d'un compteur d'itérations
error = 1  # initialisation d'une variable qui contiendra |u_np1 - u_n|
while error > 0.01:  # la boucle tourne tant que l'erreur entre deux termes successifs es
    print("itération: ", k, " ; u_n: ", u_n)  # affichage
    u_np1 = np.sqrt(u_n)  # mise à jour de la suite
    error = abs(u_np1 - u_n)  # calcul de l'erreur entre deux termes successifs
    u_n = u_np1  # mise à jour de la valeur de la suite
    k += 1  # mise à jour du compteur d'itérations
print("itération: ", k, " ; u_n: ", u_n)  # affichage
```

```
itération:  0  ; u_n:  3
itération:  1  ; u_n:  1.7320508075688772
itération:  2  ; u_n:  1.3160740129524924
itération:  3  ; u_n:  1.147202690439877
itération:  4  ; u_n:  1.0710754830729146
itération:  5  ; u_n:  1.0349277670798647
itération:  6  ; u_n:  1.0173139963058921
itération:  7  ; u_n:  1.0086198472694716
```

1.6 : Fonctions

Les **fonctions**, comme en mathématiques, permettent de renvoyer un ou plusieurs résultats en fonction d'un ou plusieurs arguments.

1.6.1 : Syntaxe et utilisation

La syntaxe est la suivante :

- mot-clé `def` pour commencer la définition d'une fonction,
- suivi du nom de la fonction (variable globale connue dans tout le programme),
- suivi du ou des arguments de la fonction (variables locales connues seulement à l'intérieur de la fonction),
- suivi de `:`,
- ensuite, dans le corps de la fonction, on va calculer le résultat voulu, et le renvoyer à l'aide du mot-clé `return`.

In [41]:

```python
def ajouter_1(x):
    return x + 1


def addition(a, b):
    return a + b


def soustraction(a, b):
    return a - b
```

Une fois qu'une fonction est définie, il faut l'appeler avec ses arguments.

In [42]:
```
print("Fonction ajouter_1 appliquée à 1 : ", ajouter_1(1))
print("Fonction addition appliquée à 1 et 2 : ", addition(1, 2))
print("Fonction soustraction appliquée à 1 et 2 : ", soustraction(1, 2))
```

```
Fonction ajouter_1 appliquée à 1 :  2
Fonction addition appliquée à 1 et 2 :  3
Fonction soustraction appliquée à 1 et 2 :  -1
```

On peut bien sûr stocker les arguments et les résultats de la fonction dans des variables.

In [47]:
```
x = 4.4
x_plus_1 = ajouter_1(x)
print("Résultat de la fonction ajouter_1 appliquée à ", x, " : ", x_plus_1)
```

```
Résultat de la fonction ajouter_1 appliquée à  4.4  :  5.4
```

Dans le corps d'une fonction, il est tout à fait possible d'utiliser le résultat de fonctions déjà définies.

Notons que les arguments d'entrée de la fonction, ainsi que les potentielles variables temporaires utilisées à l'intérieur du corps de la fonction, sont des variables **locales**, qui ne sont connues que dans la fonction.

In [52]:

```
def addition_et_soustraction(a, b):
    a_plus_b = addition(a, b)
    a_moins_b = soustraction(a, b)
    return (a_plus_b, a_moins_b)  # cette fonction renvoie un tuple


print("Fonction addition_et_soustraction appliquée à 1 et 2 : ", addition_et_soustractior

print("Valeur de la variable a_plus_b : ", a_plus_b)
# renvoie une erreur : la variable a_plus_b est locale à la fonction addition_et_soustrac
```

```
Fonction addition_et_soustraction appliquée à 1 et 2 :  (3, -1)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last
t)
<ipython-input-52-d2ec8dc80715> in <module>
      7 print("Fonction addition_et_soustraction appliquée à 1 et 2 : ", a
ddition_et_soustraction(1, 2))
      8
----> 9 print("Valeur de la variable a_plus_b : ", a_plus_b)
     10 # renvoie une erreur : la variable a_plus_b est locale à la foncti
on addition_et_soustraction

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

1.6.2 : Passage par référence des arguments d'entrée

En Python, les arguments d'entrée sont passés **par référence** aux fonctions.

Ainsi, le comportement qu'on avait observé en 1.3.1 sur les listes se retrouve lorsqu'on appelle des fonctions Python.

Lorsqu'on passe un entier en argument de la fonction, on peut le modifier localement dans la fonction, ça n'affecte pas la valeur globale de cet entier.

In [53]:

```
def func_integer(n):
    n = 2
    return n


mon_entier = 3
print("mon_entier avant l'appel de func_integer : ", mon_entier)

result = func_integer(mon_entier)

print("résultat de func_integer(mon_entier) : ", result)
print("mon_entier après l'appel de func_integer : ", mon_entier)
```

```
mon_entier avant l'appel de func_integer :  3
résultat de func_integer(mon_entier) :  2
mon_entier après l'appel de func_integer :  3
```

Ce comportement ne se retrouve pas dans le cas des listes ! La liste ci-dessous va être modifiée, car elle est passée par référence à la fonction `func_list`.

In [54]:
```
def func_list(L):
    L.append("III")
    return L


ma_liste = [1, "deux"]
print("ma_liste avant l'appel de func_list : ", ma_liste)

result = func_list(ma_liste)

print("résultat de func_list(ma_liste) : ", result)
print("ma_liste après l'appel de func_list : ", ma_liste)
```

```
ma_liste avant l'appel de func_list :  [1, 'deux']
résultat de func_list(ma_liste) :  [1, 'deux', 'III']
ma_liste après l'appel de func_list :  [1, 'deux', 'III']
```

1.7 : Conclusion

Dans ces rappels, on a introduit :

- tous les outils de base de la programmation en Python,
- les fonctions, cœur de la programmation procédurale utilisée jusqu'a présent.

Prochain cours : Introduction à la programmation orientée objet et à ses outils.

1.8 : TD

1.8.1 : Exercice 1 - Triangles

On se donne une liste `L` (de taille `3`) de listes (de taille `2`) représentant les trois sommets d'un triangle.

1) Écrire un algorithme permettant de déterminer si le triangle est isocèle, équilatéral, rectangle, ou rien de tout ça.

2) On se donne un point P (représenté par une liste de taille `2`). Ce point P appartient au triangle de sommets A, B et C si les trois expressions

$$(A-P)\wedge(B-P),\quad (B-P)\wedge(C-P),\quad (C-P)\wedge(A-P)$$

ont toutes le même signe. Ici, $\wedge$ représente le produit vectoriel :

$$(x_A,y_A)\wedge(x_B,y_B)=x_Ay_B-x_By_A.$$

Écrire l'algorithme permettant de réaliser ce test.

1) On commence par extraire les points `A`, `B` et `C` du triangle à partir de `L`. On écrit `A = L[0]`, `B = L[1]`, `C = L[2]`.

Les indices des listes en `Python` commencent à `0` !

On calcule ensuite les distances entre ces points pour avoir les longueurs des côtés ; on rappelle que la distance AB entre deux points A et B est donnée par $\sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$.

En `Python`, il faut écrire `import numpy as np` pour utiliser des fonctions mathématiques, comme la racine carrée (`np.sqrt`).

On écrit alors `AB = np.sqrt((B[0] - A[0])**2 + (B[1] - A[1])**2)`, et de même pour `AC` et `BC`.

Maintenant qu'on a les longueurs, on peut tester si un triangle est isocèle, équilatéral, rectangle :

- isocèle si `(AB == BC) OU (AB == AC) OU (AC == BC)`,
- équilatéral si `(AB == BC) ET (AB == AC) ET (AC == BC)`,
- rectangle si `(AC**2 + BC**2 == AB**2) OU (AB**2 + BC**2 == AC**2) OU (AB**2 + AC**2 == BC**2)`.

In [48]:

```
# on importe la bibliothèque numpy
import numpy as np

# on se donne une liste de listes L
L = [[0, 0], [0, 2], [2, 0]]

# on extrait les points
A = L[0]
B = L[1]
C = L[2]

# on calcule les longueurs des côtés
AB = np.sqrt((B[0] - A[0])**2 + (B[1] - A[1])**2)
AC = np.sqrt((C[0] - A[0])**2 + (C[1] - A[1])**2)
BC = np.sqrt((C[0] - B[0])**2 + (C[1] - B[1])**2)
```

In [49]:

```
# on teste si le rectangle est isocèle
est_isocèle = False
if (AB == BC) or (AB == AC) or (AC == BC):
    est_isocèle = True
```

In [50]:

```
# on teste si le rectangle est équilatéral
est_équilatéral = False
if (AB == BC) and (AB == AC) and (AC == BC):
    est_équilatéral = True

# plus rapide :
est_équilatéral = (AB == AC == BC)
```

In [53]:

```
# on teste si le triangle est rectangle
est_rectangle = False
if (AC**2 + BC**2 == AB**2) or (AB**2 + BC**2 == AC**2) or (AB**2 + AC**2 == BC**2):
    est_rectangle = True
```

In [54]:

```
# on affiche les propriétés du triangle
def affichage(est_isocèle, est_rectangle, est_équilatéral):
    if est_isocèle and est_rectangle:
        print("le triangle est rectangle isocèle")
    else:
        if est_isocèle:
            print("le triangle est isocèle")
        elif est_équilatéral:
            print("le triangle est équilatéral")
        elif est_rectangle:
            print("le triangle est rectangle")
        else:
            print("le triangle n'est ni isocèle, ni équilatéral, ni rectangle")

affichage(est_isocèle, est_rectangle, est_équilatéral)
```

```
le triangle est isocèle
```

Ici, la **précision machine** a joué un rôle ! On n'a pas l'égalité exacte entre `AB**2 + AC**2` et `BC**2`.

In [55]:
```
print("AB**2 + AC**2 == BC**2 ? ", AB**2 + AC**2 == BC**2)
print("valeur de AB**2 + AC**2 - BC**2 : ", AB**2 + AC**2 - BC**2)
print("fonction np.isclose : ", np.isclose(AB**2 + AC**2, BC**2))
```

```
AB**2 + AC**2 == BC**2 ?  False
valeur de AB**2 + AC**2 - BC**2 :  -1.7763568394002505e-15
fonction np.isclose :  True
```

In [56]:

```
est_rectangle = False
if (np.isclose(AC**2 + BC**2, AB**2) or
    np.isclose(AB**2 + BC**2, AC**2) or
    np.isclose(AB**2 + AC**2, BC**2)):
    est_rectangle = True

affichage(est_isocèle, est_rectangle, est_équilatéral)
```

```
le triangle est rectangle isocèle
```

2) On va commencer par écrire une fonction pour calculer le produit vectoriel entre deux listes `L_1` et `L_2`, représentant deux points.

On appliquera ensuite cette fonction pour calculer les produits vectoriels, et tester leurs signes. Pour cela, on créera une fonction pour calculer la différence entre deux listes.

In [57]:

```
def produit_vectoriel(L_1, L_2):
    return L_1[0] * L_2[1] - L_1[1] * L_2[0]

def moins(L_1, L_2):
    return [L_1[0] - L_2[0], L_1[1] - L_2[1]]

def est_dans_ABC(A, B, C, P):
    test_1 = produit_vectoriel(moins(A, P), moins(B, P))
    test_2 = produit_vectoriel(moins(B, P), moins(C, P))
    test_3 = produit_vectoriel(moins(C, P), moins(A, P))

    if ((test_1 >= 0 and test_2 >= 0 and test_3 >= 0) or
        (test_1 <= 0 and test_2 <= 0 and test_3 <= 0)):
        print("Le point", P, "est dans le triangle ABC.")
    else:
        print("Le point", P, "n'est pas dans le triangle ABC.")

est_dans_ABC(A, B, C, [0.5, 0.5])
est_dans_ABC(A, B, C, [0.5, 1.5])
```

```
Le point [0.5, 0.5] est dans le triangle ABC.
Le point [0.5, 1.5] est dans le triangle ABC.
```

1.8.2 : Exercice 2 - Tri

On souhaite écrire un algorithme de **tri**, l'algorithme de **tri par sélection** : étant donné un tableau `T`,

- rechercher le plus petit élément du tableau, et l'échanger avec l'élément d'indice 0 ;
- rechercher le second plus petit élément du tableau, et l'échanger avec l'élément d'indice 1 ;
- continuer de cette façon jusqu'à ce que le tableau soit entièrement trié.

Écrire cet algorithme en utilisant les listes, la structure de contrôle `if`, ainsi que des boucles.

On part d'un tableau `T`, de longueur `n`. La procédure est la suivante.

In [35]:
```
def tri_par_sélection(T):
    n = len(T)
    for i in range(n - 2):
        i_min = i
        for j in range(i + 1, n - 1):
            if T[j] < T[i_min]:
                i_min = j
        if i_min != i:
            T[i], T[i_min] = T[i_min], T[i]

T = [7, 3, 2, -1, 9, 29]
print("T = ", T)
tri_par_sélection(T)
print("T trié = ", T)
```

```
T =  [7, 3, 2, -1, 9, 29]
T trié =  [-1, 2, 3, 7, 9, 29]
```