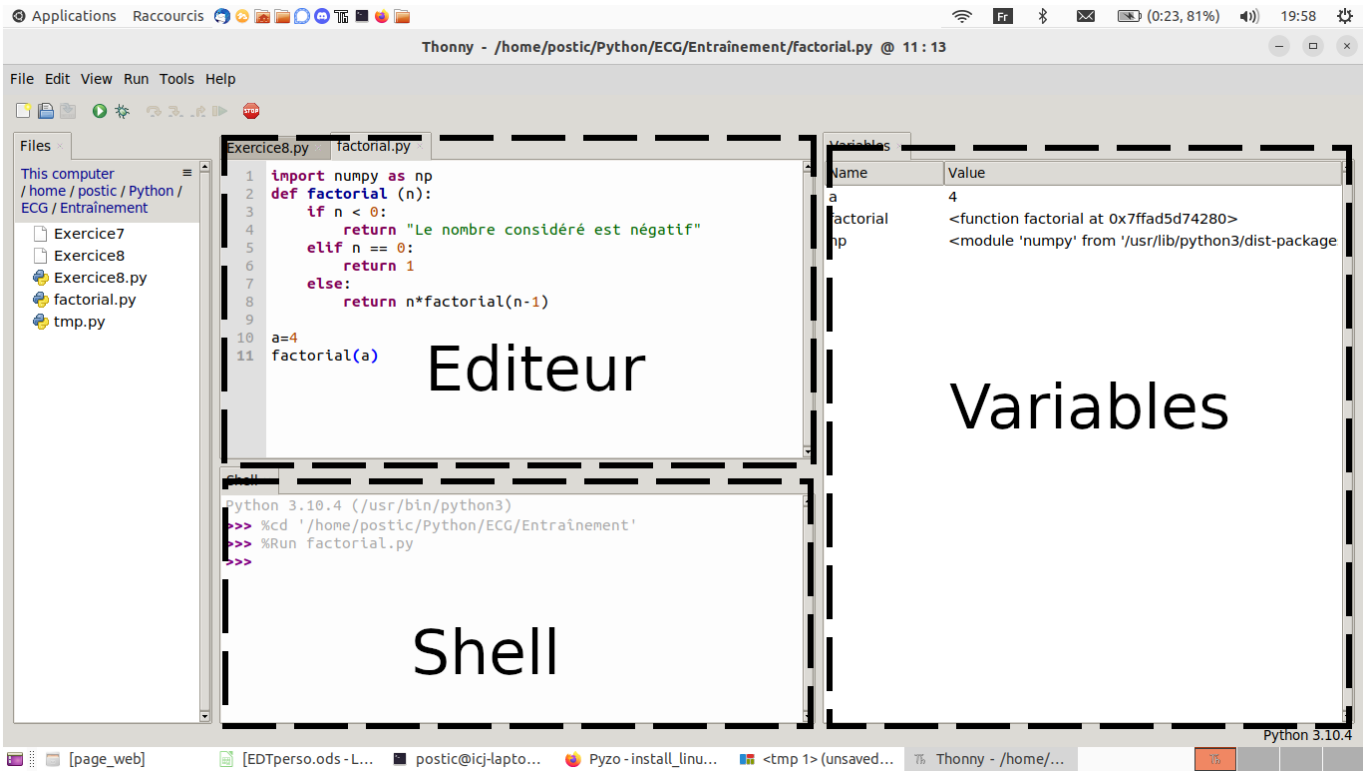
Premiers pas avec Python
TP 1

1 Le shell

1.1 L'interface :

Thonny est un environnement de développement intégré (abrégé IDE en anglais) qui propose une interface en plusieurs morceaux. On travaillera le plus souvent avec la configuration suivante :



1. **Le shell :** Le shell, ou la console, est la partie "calculatrice" de Thonny. C'est dans le shell que nous allons faire les calculs, lancer les programmes créés avec l'éditeur, exécuter des instructions, etc.
2. **L'éditeur :** L'éditeur permet d'écrire des séries d'instructions, enregistrées dans des fichiers, qu'on appelle des "feuilles de code". C'est dans l'éditeur que l'on crée nos programmes, qui seront ensuite lancés depuis le shell après qu'on les aura "exécutés". On essaie en général de faire tenir tous les programmes nécessaires à un projet dans une seule feuille.
3. **Les variables :** Permet de voir ce que Python a actuellement en mémoire. Ici, la mémoire contient 3 éléments, un nombre, une fonction, et un module. On reviendra sur ces différents éléments par la suite.
4. **L'explorateur de fichier :** Permet de naviguer directement vers vos fichiers.

Avant de commencer : Pour faire apparaître les variables, aller dans View et cocher Variables. Vous pouvez aussi, une fois votre fichier ouvert ou créé, enlever la partie Files (pour la remettre, View -> Files).

1.2 Calculatrice

Commençons par utiliser Python comme calculatrice en testant quelques commandes....

Exercice 1 Écrire chacune des instructions suivantes dans le shell en appuyant à la fin de chaque ligne sur "Entrée" et en prenant le temps d'observer et de comprendre ce que cela fait. Si vous avez un message d'erreur, interprétez. N'hésitez pas à essayer d'autres valeurs si vous ne voyez pas ce que cela fait... Puis marquez sur votre cahier de Python ce que font ces premières commandes.

dans le shell
```
2+3
4*2
4+5*2
4+(5*2)
(4+5)*2
3/2
3*(-2)
3*-2
3**2
7%3
10//3
```

Remarques :

1. Pour reprendre automatiquement une ligne écrite précédemment, vous pouvez appuyer sur la flèche du haut autant de fois que nécessaire (ou flèche du bas pour redescendre)
2. La puissance s'exprime avec un double symbole *, et non pas avec ^. Ceci constitue une différence avec les langages les plus fréquents.

Exercice 2 Quand vous lancez Thonny, beaucoup de fonctions classiques en maths ne sont pas reconnues :

1. Essayez les instructions suivantes dans le shell :

   dans le shell
   ```
   cos(0)
   pi
   ```

2. Essayez maintenant ceci :

   dans le shell
   ```
   from numpy import *
   cos(0)
   pi
   exp(1)
   ```

Vous venez d'importer le module "numpy" (pour "numeric python") qui ajoute les fonctions mathématiques et certaines variables. Dans la partie Variables, vous pouvez constater l'apparition de tout un ensemble de nouvelles variables, la plupart étant des fonctions. Ce sont toutes les fonctions déjà codées enregistrées dans la bibliothèque numpy.

La plupart de nos programmes auront besoin de modules pour fonctionner, et il y a deux plusieurs façons de les utiliser :

Si on veut tout importer :

```
from nomdumodule import *
```

le "*" signifie que vous importez "tout" le module. Si vous vouliez importer qu'une seule fonction de numpy, par exemple la racine carré, on écrirait :

```
from  numpy import sqrt
```

Exercice 3 (Reinitialisation) Dans la barre de menu de Thonny, cliquez sur "Run", puis sur "Stop/Restart backend". Pour plus de clarté, appuyez aussi sur Ctrl + L, cela videra la console. Exécutez maintenant la commande suivante :

dans le shell
```
pi
```

Vous venez de réinitialiser la console et avez donc effacé de la mémoire de Thonny les fonctions mathématiques (et tout ce que vous aviez fait avant...) comme vous pouvez le voir dans la partie Variables.

On peut importer d'une dernière façon, souvent utilisée pour mieux repérer les modules importés :

```
import numpy as np
```

Cette instruction fait que "np" signifie maintenant "numpy" et permet d'aller chercher les fonctions dans cette banque.

Essayez les instructions suivantes :

Exercice 4 (module dans un raccourci) Réinitialisez la console comme on vient de le voir (vous pouvez aussi utiliser Ctrl+F2). Exécutez maintenant les commandes suivantes :

dans le shell
```
import numpy as np
sqrt(4)
np.sqrt(4)
```

1.3 Les variables et affectations

Exercice 5 En Python, on peut enregistrer dans une lettre (ou un mot) un contenu. Cette opération s'appelle "l'affectation", et le contenu est stocké dans une "variable". L'affectation se fait avec le symbole "=" dans le même ordre qu'en maths (c'est à dire `variable=contenu`). Essayez les instructions suivantes :

dans le shell
```
a=2
a
3+a
3*a
3a
```

Vous avez enregistré le nombre 2 dans la variable a. On dit que vous avez "affecté" à a la valeur 2. **Notez que vous ne pouvez pas utiliser le produit implicite** : le signe $*$ est obligatoire pour le produit.

Exercice 6 Devinez AVANT de les taper l'effet des instructions suivantes et vérifiez ensuite en les testant dans le shell :

dans le shell
```
a=2
b=3*a
b
a=4
b
a=a+1
a
a=a+1
a
```

Remarques :

1. On peut affecter une variable avec une autre variable à condition que cette dernière soit déjà affectée (cf la ligne `b=3*a` ci dessus, possible car `a` a été affecté à la ligne précédente. Dans le cas contraire, on aura une erreur "variable non définie".
2. L'instruction `a=a+1` s'appelle une incrémentation.

Une variable peut contenir n'importe quel **type** d'objet. Nous avons pour l'instant uniquement vu des nombres (des entiers, et des nombres réels non entiers), mais nous verrons plusieurs autres types cette année :

— entier `int` pour `integer`, réel `float`
— matrice ou tableau `array`
— chaîne de caractère `str`. Une chaîne de caractères est une suite de symboles entre ' ' ou " ". Par exemple `"Bonjour"` ou `'$!3m'`.

Exercice 7 Affectez deux variables de la manière suivante : `a="chocolat"` et `b="caramel"`.

Proposez maintenant une série d'instructions, avec uniquement des affectations, qui intervertit le contenu de a et de b. Ainsi, on veut qu'à l'issue de la série d'opérations, a contienne le mot "caramel" et b le mot "chocolat". Evidemment, on s'interdit d'écrire `a="caramel"` et `b="chocolat"`.

Notez sur votre cahier de Python cette technique dont on se resservira souvent.

Pour intervertir le contenu de a et de b, on peut utiliser une variable temporaire `temp` pour stocker l'un des deux contenus pendant l'échange. Voici une série d'instructions qui réalise cette opération :

dans l'éditeur
```
a="chocolat"
b="caramel"
temp=a
a=b
b=temp
```

Après l'exécution de ces instructions, la variable `a` contiendra "caramel" et la variable `b` contiendra "chocolat".

2 L'éditeur

2.1 Un premier script

Naviguer dans les lignes du "shell" n'est pas très pratique, et si on veut effectuer une série d'opérations, cela peut s'avérer très fastidieux. En particulier, on **ne peut pas** effectuer une opération dans le shell et la modifier a posteriori !

Heureusement, on dispose d'un éditeur qui permet de créer des "scripts", ou des "programmes" dans lesquels écrire nos opérations.

Considérons le polynôme $P(x) = 3x^2 + 4x + 1$. On cherche à évaluer P pour diverses valeurs. On va pour cela utiliser un script.

Tout se passe dans la partie gauche de votre fenêtre, en dessous de ce qui est noté `<untitled>`. Écrivez dans cette zone :

dans l'éditeur
```
x=1
P=3*x**2+4*x+1
print(P)
```

On va maintenant "exécuter" le script. Pour cela, cliquez sur le bouton "Play" ou "Run -> Run current script" ou utilisez le raccourci F5. Dans un premier temps, Thonny va vous demander d'enregistrer le script, faites-le (appelons le tmp.py).

Observez ensuite ce qui se passe dans le shell (en dessous donc) : vous devriez voir écrit `(%Run tmp.py)`, puis un '8' en dessous. C'est la valeur de P.

Si vous voulez calculer $P(8)$, il suffit de remplacer la première ligne du script par `x=8`. Essayez !

Remarquez maintenant la dernière ligne avec le `print`. Elle est nécessaire dans les scripts si vous voulez "afficher un résultat", mais les calculs se font même si vous l'oubliez.

2.2 Création de fonctions

On peut faire bien plus pratique que modifier à chaque fois le script en créant des fonctions qui se comportent (presque) comme des fonctions mathématiques.

Dans la fenêtre d'édition, écrire les commandes suivantes en respectant SCRUPULEUSEMENT le décalage proposé (Thonny devrait le faire automatiquement, mais si besoin vous obtiendrez le bon décalage avec la touche TAB ($\rightleftharpoons$) du clavier) :

dans l'éditeur
```
def P(x):
    return 3*x**2+4*x+1
```

La variable `x` de la fonction est appelée "argument" de la fonction.

Remarquez que cette définition est semblable à celle qu'on aurait en maths $P : x \mapsto 3x^2 + 4x + 1$.

Le `return` correspond à ce qu'on a à droite de la flèche de définition, et à gauche on a le nom de la fonction et l'argument, ici `P` et `x`.

Exécutez le script (raccourci F5 ou passez par les menus) et tapez `P(1)` **dans le shell**, puis `P(-1)`.

Vous devriez voir afficher les résultats du calcul donnant $P(1)$ et $P(-1)$.

Ce qu'il se passe pour python, c'est que `x` est remplacé par 1 (puis -1) dans le programme, le calcul est effectué, et le résultat est "retourné" ou "renvoyé" dans la console. Là encore, tout se passe comme quand vous calculez `P(1)` et `P(-1)` en maths.

Exercice 8 Écrire de cette façon les fonctions $x \mapsto x^3 + 2x - 5$, $x \mapsto \exp(x+3)$, $x \mapsto \sin(x) + \cos(x^2)$ et les tester pour différentes valeurs.

Attention : n'oubliez pas d'inclure le module numpy au début du script !

Voici comment vous pouvez définir les fonctions demandées dans l'éditeur de Thonny :

dans l'éditeur
```
from numpy import *
def f1(x):
    return x**3 + 2*x - 5
def f2(x):
    return exp(x + 3)
def f3(x):
    return sin(x) + cos(x**2)
```

Après avoir exécuté ce script (F5), vous pouvez tester les fonctions dans le shell comme suit :

dans le shell
```
f1(2)  # Calcule 2^3 + 2*2 - 5 = 8 + 4 - 5 = 7
f2(-3)  # Calcule exp(-3 + 3) = exp(0) = 1
f3(0)  # Calcule sin(0) + cos(0^2) = 0 + 1 = 1
```

Vous pouvez remplacer les arguments par d'autres valeurs pour tester les fonctions davantage. La partie `# ...` dans le shell est un commentaire pour expliquer ce que fait chaque ligne. Elle n'est pas utilisée par Python.

Remarque : Une autre façon d'écrire des fonctions en Python est d'utiliser la syntaxe `input`. Je vous la présente ici, mais nous l'utiliserons beaucoup moins (elle est en fait réservée à des interactions avec un utilisateur, ce qui est beaucoup moins fréquent que les cas d'utilisation d'une fonction classique)

Dans l'éditeur écrivez

dans l'éditeur
```
print('P(x)=3*x**2+4*x+1')
n=float(input('pour quel x voulez vous calculer P(x)?'))
a=3*n*n+4*n+1
print(a)
```

Cette syntaxe est assez lourde, car il faut expliciter le type de l'input (compris par la machine comme une chaîne de caractères).

3 Algorithme et programmation : if et while

3.1 le bloc if

On a très vite besoin d'instructions plus compliquées que juste des calculs et des affectations. Prenons par exemple la fonction valeur absolue $x \mapsto |x|$.

On rappelle que cette application absolue est définie par :

$$|\ |\ : \begin{array}{ccc} \mathbb{R} & \to & \mathbb{R} \\ x & \mapsto & \begin{cases} x & \text{si } x \geq 0 \\ -x & \text{si } x < 0 \end{cases} \end{array}$$

Pour cette fonction, on a donc besoin de tester si x est plus grand que 0. Python permet de gérer cela très simplement avec l'instruction "if". Recopiez l'exemple suivant dans l'éditeur (là encore, Thonny gère normalement tout seul les espaces, mais faites bien attention à respecter scrupuleusement ces indentations) :

dans l'éditeur
```
def valeurabs(x):
    if x >= 0:
        return x
    else:
        return -x
```

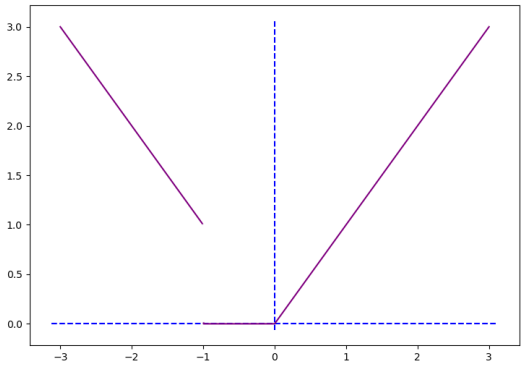
Et testez maintenant dans la console `valeurabs(3)`, `valeurabs(-2)`.

Vous avez bien créé une nouvelle fonction, appelée `valeurabs` et qui se comporte comme la valeur absolue en maths.

On peut également ajouter des conditions supplémentaires avec l'instruction `elif` :

dans l'éditeur
```
def valabs(x):
    if x > 0:
        return x
    elif -1<x<0 :
        return 0
    else:
        return -x
```

Ceci crée la fonction dont voici le graph :



3.2 Opérateurs logiques

Pour écrire vos conditions, vous aurez besoin d'utiliser les instructions suivantes :

Instruction	*Effet*
`==`	égal
`>`	$>$
`<`	$<$

Instruction	*Effet*
`<=`	$\leq$
`>=`	$\geq$
`!=`	$\neq$

Instruction	*Effet*
`or`	ou
`and`	et
`not`	négation

Marquez les dans votre cahier de Python, on s'en resservira toute l'année !

Remarque : Vous pouvez remarquer que Python traite différemment le `=` et le `==`. Il faudra y faire attention par la suite !

3.3 Exercices

Exercice 9 Créer la fonction suivante f définie par :

$$f(x) = \begin{cases} x^2 & \text{si } x > 1 \\ x & \text{si } -1 \leq x \leq 1 \\ -x^3 & \text{si } x < -1 \end{cases}$$

dans l'éditeur
```
def f(x):
    if x>1:
        return x**2
    elif -1<=x<=1:
        return x
    else:
        return -x**3
```

Vous pouvez tester cette fonction dans le shell avec différentes valeurs de x pour vérifier son comportement.

dans le shell
```
f(2)    # Devrait retourner 4
f(0.5)  # Devrait retourner 0.5
f(-2)   # Devrait retourner 8
```

Exercice 10
Écrire une fonction que vous appellerez `maxi`, qui prend deux variables a et b et qui retourne la plus grande des deux.

Par exemple `maxi(-2,3)` retournera 3, ou `maxi(4,4)` devra retourner 4.

dans l'éditeur
```
def maxi(a,b):
    if a>=b:
        return a
    else:
        return b
```

Vous pouvez tester cette fonction dans le shell avec différentes paires de valeurs pour vérifier son comportement.

dans le shell
```
maxi(-2,3)  # Devrait retourner 3
maxi(4,4)   # Devrait retourner 4
maxi(0,-1)  # Devrait retourner 0
```

Exercice 11

1. Écrire une fonction qui admet trois variables a, b et c et qui donne le nombre de racines réelles d'un polynôme $P(X) = aX^2 + bX + c$.
2. Même question mais qui donne la valeur de ces racines (toujours uniquement pour le cas réel)

1. dans l'éditeur
```
def nombre_de_racines(a,b,c):
    D=b**2-4*a*c
    if D>0:
        return 2
    elif D==0:
        return 1
    else:
        return 0
```

Vous pouvez tester cette fonction dans le shell avec différentes valeurs de a, b et c pour vérifier son comportement.

2. dans l'éditeur
```
def racines_reelles(a,b,c):
    D=b**2-4*a*c
    if D>0:
        r1=(-b+(D)**0.5)/(2*a)
        r2=(-b-(D)**0.5)/(2*a)
        return ('Le polynôme a deux racines qui sont :', r1, r2)
    elif D==0:
        r=-b/(2*a)
        return ('Le polynôme a une racine double qui est :', r)
    else:
        return ('Le polynôme n\'a pas de racines réelles.')
```

3.4 Boucle while :

Soit un entier $N \in \mathbb{N}$. On cherche à partir de quel entier n on a $2^n \geq N$.
Comment procéder sans résoudre d'équation ?

Une idée est d'y aller "à tâtons" : on regarde si ça marche avec 2^1, puis avec 2^2, puis 2^3, etc... Une fois qu'on a dépassé la quantité N, on s'arrête.

C'est ce que va permettre de faire l'instruction `while` dont la syntaxe est la suivante :

dans l'éditeur
```
while condition:
    instruction
    instruction
    instruction
suite du programme
```

Tant que ce que vous avez mis à la place de `condition` est vérifié, les instructions qui sont "indentées" (c'est à dire décalées) sont exécutées. Lorsque la condition n'est plus vérifiée, on passe à la suite du programme (qui n'est pas indentée).

Exercice 12
Écrire un programme dont l'argument s'appelle N et qui répond le plus petit n tel que $2^n \geq N$.

dans l'éditeur
```
def petit_n(N):
    n=0
    puissance=1  # Ceci correspond à 2**n
    while puissance < N:
        n=n+1
        puissance=2*puissance
    return n
```

Ici, plutôt que de recalculer 2^n à chaque étape, on le met à jour en le multipliant par 2 : `puissance` contient donc 1, puis 2, puis 4, puis 8, etc...

Exercice 13
Créer des programmes utilisant la boucle while pour calculer les sommes suivantes :

1. $\sum_{k=1}^{n} k$: la première ligne du programme sera `def somme(n)` :
2. $\sum_{k=0}^{n} q^k$: la première ligne du programme sera `def geom(q,n)` :
3. Trouver le plus petit n tel que $\sum_{k=0}^{n} q^k > 10^6$.

On comparera à chaque fois avec les formules du cours.

1. dans l'éditeur
```
def somme(n):
    total=0
    k=1
    while k <= n:
        total=total + k
        k=k+1
    return total
```

2. dans l'éditeur
```
def geom(q,n):
    total=0
    k=0
    while k <= n:
        total=total + q**k
        k=k+1
    return total
```

3. dans l'éditeur
```
def petit_n_geom(q):
    total=0
    k=0
    while total <= 10**6:
        total=total + q**k
        k=k+1
    return k-1  # On retourne k-1 car k a été incrémenté une derni
```

Je vous laisse le soin de comparer les résultats obtenus avec les formules du cours !

Exercice 14

1. Écrire un programme qui calcule $n!$ et le réutiliser pour calculer $\binom{n}{k}$ avec la première formule du cours.
2. Utiliser ce programme pour calculer $\binom{10000000000000000}{1}$
3. Écrire un programme utilisant la deuxième formule pour $\binom{n}{k}$. Comparez l'efficacité des deux programmes en calculant $\binom{10000000000000000}{1}$ avec ce deuxième programme

1.
```
dans l'éditeur
def factorielle(n):
    resultat=1
    k=1
    while k <= n:
        resultat=resultat * k
        k=k+1
    return resultat
def binomiale1(n,k):
    return factorielle(n) // (factorielle(k) * factorielle(n - k))
```

2. Le calcul de $\binom{10000000000000000}{1}$ avec ce programme est très lent et peut même ne pas aboutir en raison de la taille énorme des factorielles impliquées.
3. On utilise ici la formule

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k!}$$

```
dans l'éditeur
def binomiale2(n,k):
    if k > n:
        return 0
    elif k == 0 or k == n:
        return 1
    else:
        numerateur=1
        i=0
        while i < k:
            numerateur=numerateur * (n - i)
            i=i+1
        return numerateur // factorielle(k)
```

Ce programme est beaucoup plus efficace pour calculer $\binom{10000000000000000}{1}$, car il évite le calcul des factorielles complètes et utilise une approche itérative.