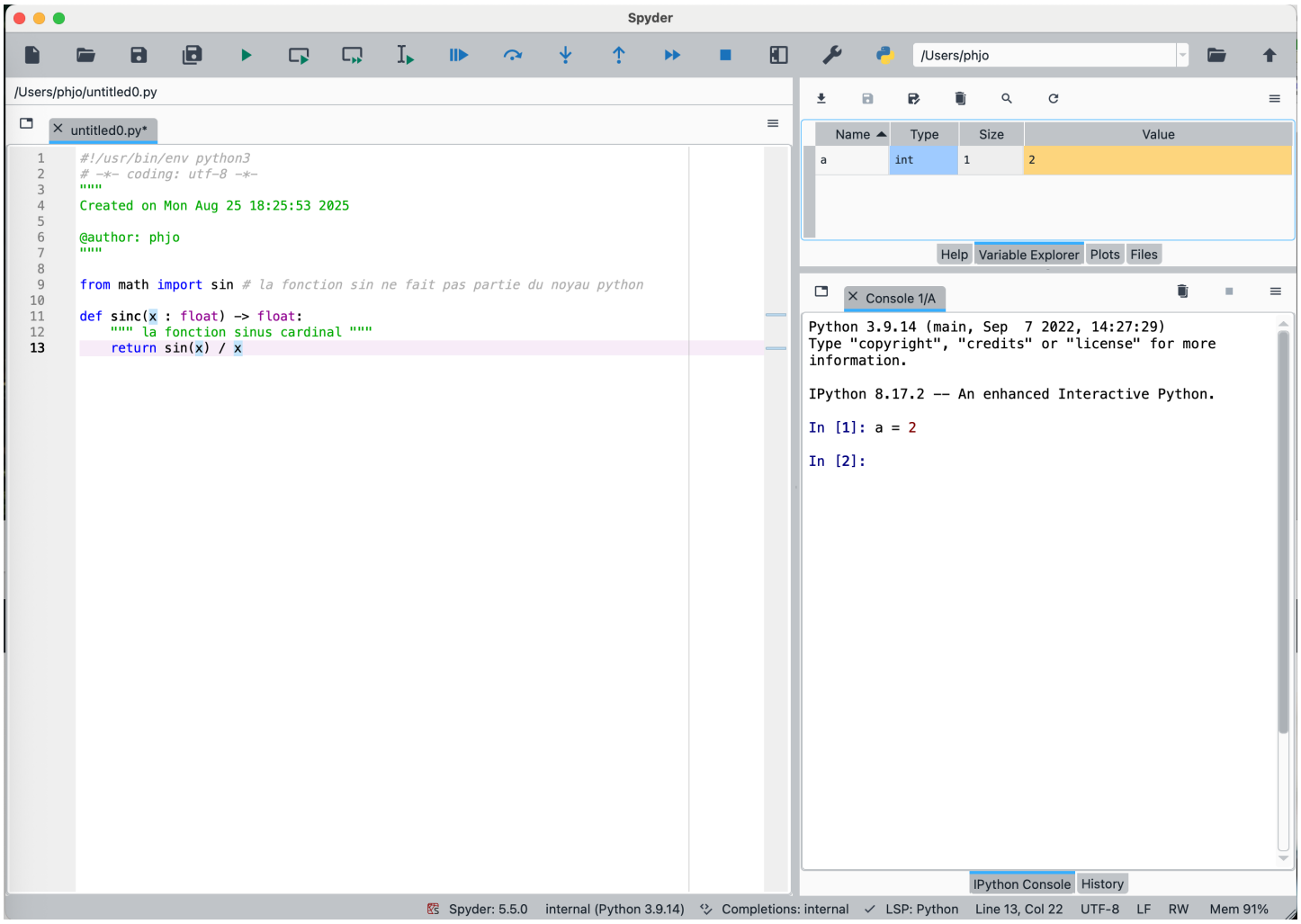
TP 1 - prise en main de python et spyder

Dès aujourd'hui, on va travailler dans l'environnement de développement Spyder. Sur les ordinateurs du lycée, vous trouverez ce programme dans le dossier Mathématiques->WinPython. Lancez donc sans plus attendre Spyder et continuez votre lecture.

Si vous disposez déjà de vos identifiants, alors une fois connecté sur votre session, je vous invite à créer un dossier `InfoPCSI` puis un sous-dossier `TP1` où vous enregistrerez le fichier relatif à ce premier TP.

Voici à quoi ressemble Spyder sur mon ordinateur :



La fenêtre de gauche est un éditeur, où on peut écrire des programmes (lesquels peuvent être constitués d'un ou de plusieurs fichiers) et la fenêtre en bas à droite est la console, où on peut exécuter en direct des commandes Python, tester des parties de code qu'on a écrites dans l'éditeur. En haut à droite, la dernière fenêtre dispose de plusieurs onglets vous permettant de connaître les valeurs de variables, consulter de la documentation sur des commandes python, visualiser des sorties graphiques ou les fichiers du répertoire courant.

Un point important à retenir est que, dans la console, les commandes sont exécutées immédiatement, une par une, tandis que celles figurant dans l'éditeur ne le sont qu'en bloc, et ce lorsqu'on le décide. Une erreur classique consiste alors à oublier de réexécuter le code écrit dans l'éditeur après quelques changements, ce qui conduit souvent à des moments de perplexité lorsque les commandes exécutées dans la console et faisant appel à telle ou telle fonction de notre code ont un résultat incompatible avec le code que l'on a sous les yeux.

Moralité : mieux vaut enregistrer/exécuter souvent, d'autant qu'une fausse manœuvre est vite arrivée ! (ou un plantage de Spyder, ce qui conduit à perdre les parties non encore sauvegardées de notre travail...)

1 Opérations sur des entiers (int)

Les opérateurs qui agissent sur les entiers sont `+, -, *, //, **, %`. Faites dans la console les calculs suivants : `1+2*3`, `2-3+4`, `5//2`, `2**5`, `2**2**3`, `(2**2)**3`, `2**(2**3)`, `5%2`.

Question 1. En supposant que `a` et `b` désignent deux entiers positifs (`b` non nul), qu'obtient-on avec l'expression `a // b` ? et avec `a % b` ? (N'hésitez pas à faire d'autres tests pour confirmer votre intuition si nécessaire)

Question 2. Classer les opérateurs cités ci-dessus par ordre croissant de précédence (vous noterez par exemple que l'opérateur de multiplication `*` est de plus haut degré de précédence que l'addition `+` ou la soustraction `-`, ce qui conduit à ce que l'expression `1+2*3` soit évaluée en `7` et non en `9`.)

Question 3. Une fois connues les précédences relatives des opérateurs d'une expression, son évaluation suit alors des règles simples : les opérateurs de plus haute précédence sont évalués en premier, et si plusieurs opérateurs de même précédence se suivent, l'évaluation se fait alors de gauche à droite (exemple : `2-3+4`).

Il y a toutefois une exception à cette règle… Laquelle ? et pourriez-vous expliquer pourquoi ?

Vous noterez que python est capable de calculer avec de grands entiers. Essayez par exemple `5**5**5`. Pour ce faire, Python est capable d'adapter la taille en mémoire qu'occupe un entier à celui-ci. On s'en doute toutefois, plus les entiers manipulés sont grands, plus les opérations arithmétiques sont lentes…

2 Nombres à virgule (float)

Le séparateur décimal en python est le point : `.` et dès lors qu'un nombre fait apparaître celui-ci, alors l'objet créé par Python n'est plus un entier (`int`) mais un nombre dit flottant (`float`). Une autre manière de faire apparaître un flottant est de diviser deux entiers avec l'opérateur `/` (et non plus `//`) ou de faire apparaître une puissance négative, ainsi `1/3`, `2**(-2)` s'évaluent en des flottants (tester !)

Contrairement aux entiers, les nombres flottants en python ne peuvent être arbitrairement grands, car la place en mémoire qu'occupe un nombre flottant est toujours la même (cela ne vous parle sans doute pas aujourd'hui, mais vous apprendrez qu'un nombre en virgule flottante est codé sur 64 bits, ou chiffres binaires, c-à-d que la représentation d'un flottant ne permet, dans le meilleur des cas, que de représenter 2^{64} nombres différents)

Les conséquences sont que :

- Après un calcul en virgule flottante, un nombre peut s'avérer trop grand pour être représentable, ce qui conduira à une erreur. (Essayez `2.0**1024` qui déclenche une erreur de type `OverflowError`)
- Un nombre réel peut s'avérer être si petit qu'il est confondu avec 0… (essayez `2.0**(-1075)`)
- Si certains nombres réels sont exactement représentables, la plupart ne le sont pas de manière exacte, mais que de manière approchée. De plus, comme l'ordinateur utilise pour ses représentations la base 2, certains nombres pourtant très simples tels que `0.1` et `0.2` ne sont pas exactement représentés. Essayez de calculer : `0.3-0.1-0.2`

Les opérateurs qui agissent sur les flottants sont `+, -, *, /, //, **, %`.

Question 4. Testez les opérateurs `//` et `%` avec des nombres en virgule flottante positifs, et exprimer d'une phrase ce que l'on obtient avec `a // b`, `a % b` étant donnés des réels strictement positifs `a` et `b`.

A noter que si dans un calcul figure un flottant et un ou des entiers, le résultat sera toujours un flottant, même si tout laisse à penser que le résultat devrait être un entier

Exemples : `0.1*10`, `1.1-0.1`. Même l'opérateur `//` dont par définition la valeur de retour est toujours un nombre entier renvoie un nombre en virgule flottante si l'un de ses opérandes est un flottant.

2.1 Module `math`

Parmi les fonctions mathématiques que l'on connaît (ou dont vous apprendrez l'existence cette année), bien peu figurent dans le noyau de Python. Une exception est la fonction valeur absolue qui s'écrit `abs` en python. Lorsqu'on a besoin de fonctions telles que $\sin, \cos, \exp, \log$ voire $\sqrt{.}$, alors il faut importer lesdites fonctions. On verra qu'il existe d'autres possibilités, mais en attendant on peut faire appel au module `math` qui définit toutes ces fonctions (et d'autres encore !) :

Pour importer ce module, on peut procéder ainsi (essayez bien sûr les commandes qui suivent dans la console) :

```
import math
help(math) # détaille toutes les fonctions du module
math.sin(math.pi/4)-math.sqrt(2)/2
```

A noter que Spyder propose une autocomplétion : tapez `math.` puis tapez encore la touche de tabulation, et vous verrez apparaître un menu à parcourir de toutes les fonctions que contient le module (pourvu que celui-ci ait été imporé avec `import math` comme ci-dessus).

Comme vous le voyez, les fonctions (et constantes) du module `math` doivent être préfixées du nom `math` pour pouvoir être utilisées, ce qui peut paraître lourd. On peut utiliser un alias pour racourcir les écritures :

```
import math as m
m.sin(m.pi/3)
```

ou bien on peut faire :

```
from math import sin, pi
sin(pi/3)
```

ou, pour les grands paresseux (cette pratique est, de manière générale, déconseillée) :

```
from math import * # on rajoute toutes les fonctions du module math...
factorial(500)
```

3 Tests, Booléens

On aura très souvent besoin de réaliser des tests : comparer deux nombres, voir si deux objets sont identiques, vérifier l'appartenance d'un objet à une collection etc... Le résultat d'un test est, on s'en doute, l'une ou l'autre des deux valeurs : vrai ou faux, ce qui sans surprise en python s'écrit `True` ou `False` (oui, avec une majuscule : python est sensible à la casse) qui sont les deux valeurs booléennes.

Parmi les opérateurs ayant une valeur de retour booléenne figurent : `<, >, <=, >=, ==, !=`

`==` est le test d'égalité, tandis que `!=` en est l'exacte négation. Attention au piège : on verra que le symbole `=` seul a une toute autre signification, et il faudra être très soigneux pour ne pas utiliser l'un à la place de l'autre...

On peut combiner plusieurs tests en un à l'aide des opérateurs booléens `and, or` et `not` ou bien, directement ainsi :

```
1 < 2 < 5 # donne True, comme on le devine
1 < 4 < 6 > 5 # donnne aussi True
```

A noter que `1 < 5 > 2` est équivalent à `(1 < 5) and (5 > 2)` ou encore à `1 < 5 and 5 > 2` (`and` est de précédence inférieure à `<` et `>` si bien qu'on peut se passer de parenthèses)

A noter que le test d'égalité `==` n'a guère de sens pour comparer deux nombres en virgule flottante. Comme ceux-ci représentent des nombres réels de manière approchée, ou sont le résultat de petites approximations dans des calculs, le seul test qui prend du sens est d'essayer de voir si deux nombres en virgule flottante sont proches l'un de l'autre ou non...

4 Variables, affectation

On peut souhaiter nommer une valeur (ou n'importe quel objet Python) pour s'y référer plus tard. C'est le rôle de l'affectation :

```
unEntier = 5**5**5
```

Voici ce qui se passe en exécutant la ligne ci-dessus : la valeur `5**5**5` est, dans un premier temps, calculée (d'abord l'entier `5**5`, à savoir 3125, puis `5**3125`). Le (grand) entier obtenu se trouve quelque part dans la mémoire de l'ordinateur, et le rôle de l'affectation va être de donner un nom à cette valeur calculée (sans qu'elle soit recalculée, bien sûr).

Si on venait à changer par la suite la valeur associée à la variable `unEntier`, par exemple avec une seconde affectation telle que `unEntier = -2542`, alors la place mémoire qu'occupait l'entier calculé `5**5**5` serait libérée, l'entier `-2542` prendrait place quelque part en mémoire (très certainement ailleurs, d'autant qu'il va occuper nettement moins de place…) et on ferait alors pointer l'étiquette `unEntier` vers ce nouvel entier.

A retenir donc : réaliser une affectation, c'est juste nommer un objet existant, pour s'y référer plus tard.

Si on exécute l'instruction suivante : `étiquette1 = étiquette2 = 6**6**6` alors la valeur `6**6**6` est calculée (une fois) et deux étiquettes viennent alors pointer vers cette même valeur (on devine qu'elle prend de la place en mémoire, car c'est tout de même en grand entier, mais elle n'est au fond, du fait de l'instruction précédente, qu'une seule fois présente en mémoire)

Bien comprendre ce principe, que plusieurs étiquettes puissent au fond représenter le même objet, est essentiel en python. On verra en effet qu'il existe des objets dits mutables, et si plusieurs étiquettes référencent un même objet mutable, toute modification de celui-ci par le biais d'une des deux étiquettes modifie évidemment aussi l'objet que référence l'autre étiquette, puisque c'est le même !

Question 5. L'affectation simultanée permet de définir les valeurs associées à plusiers variables simultanément. Par exemple : `a, b = 3, 2` réalise la même chose que les deux affectations : `a = 3` suivie de `b = 2`.

L'affectation simultanée permet de réaliser l'échange des valeurs référencées par `a` et `b` de manière très simple : il suffit d'exécuter `a, b = b, a`.

Comment faire pour réaliser cet échange sans utiliser l'affectation simultanée ? (N.B. : une solution est d'introduire une troisième variable, mais dans le cas où `a` et `b` ont des valeurs numériques telles que des entiers, on peut aussi s'en sortir sans introduire de troisième variable...)

5 Chaînes de caractères

Si le langage Python est, comme tout langage de programmation, capable de réaliser des calculs et de manipuler des nombres, il permet également de manipuler du texte, sous la forme de chaînes de caractères. Pour introduire une chaîne de caractères, il suffit de l'écrire, entourée de délimiteurs, lesquels peuvent être l'apostrophe `'` ou les guillemets `"` :

```
s1 = "Hello, world !"
print(s1)
s2 = 'Hello, world !'
s1 == s2
```

Il n'y a pas de différence donc entre les deux syntaxes, et la raison de préférer l'une à l'autre provient quelquefois de ce que la chaîne à introduire contienne une apostrophe, ou un guillemet, car si l'apostrophe est utilisée pour délimiter une chaîne, on devine que si une apostrophe est dans celle-ci, elle va marquer la fin de la chaîne à définir ! Lorsque cela est nécessaire, on peut préfixer l'apostrophe ou le guillement d'un caractère antislash \ (backslash en anglais) et par exemple écrire :

```
s3 = "Hello, \"World\" !"
```

(Si c'est le caractère \ qu'on veut faire apparaître dans une chaîne de caractères, on le doublera : \\)

Enfin, quand on a besoin d'écrire une chaîne de caractères qui s'étend sur plusieurs lignes, on la délimitera par ou bien trois apostrophes : `'''` ou bien trois guillemets : `"""`.

```
s = """ligne 1 d'une longue chaîne de caractères
...: la ligne 2
...: la ligne 3"""
s
print(s)
```

Bien sûr, recopier les exemples ci-dessus. Vous noterez la présence de deux fois la séquence `\n` qui représente le caractère de contrôle indiquant la fin d'une ligne de texte (on pourrait d'ailleurs introduire cette séquence dans une chaîne de caractères et son affichage conduirait à un saut de ligne).

Quelques commandes utiles sur les chaînes de caractères :

```
len(s1) # len est un diminutif de length = longueur en anglais...
s1[0], s1[1] # extraction des deux premières lettres de s1 : Python numérote à partir de 0
s1[0:5] # un slice : extraction des lettres de l'indice␣0␣à␣l'indice 5 (non compris)
s1[5:] # tout à partir de l'indice␣5␣(compris)
s1[:5] # tout jusqu'à␣l'indice 5 (non compris)
s1[0:7:2] # essayez ! 2 est le pas...
s1[::-1] # idem
"Hello"+'world' # + avec les chaînes de caractères est l'opérateur␣de␣concaténation
"Aïe␣"*4 # 4*'Aïe␣' marche aussi !
```

Question 6. Déterminer `a,b,c` de telle sorte que `s1[a:b:c]` s'évalue en `'dlrow'`

Remarque 1. Une chaîne de caractères est un objet immuable en python. On ne modifie donc pas une chaîne de caractères (exemple : `s1[0] = 'a'` conduit à une erreur et ne permet donc pas de remplacer le premier caractère de `s1`) mais on peut d'une chaîne de caractères en obtenir une autre en en extrayant une partie (voir les slices ci-dessus), obtenir de deux chaînes la chaîne qui les assemble en une (concaténation) etc...

6 Un premier programme

Dans l'éditeur (fenêtre de gauche), introduisez les lignes suivantes, et sauvegardez votre script, de préférence dans `InfoPCSI->TP1`. Bien sûr, si votre compte est temporaire, le répertoire choisi est de peu d'importance...

Ce qui sur une ligne suit le caractère `#` est, vous l'avez deviné, un commentaire. Vous pouvez ici le recopier, ou vous abstenir, au choix.

```
s = input("Entrez␣la␣température␣en␣degrés␣Fahrenheit␣:␣") # on obtiendra une chaîne de
caractères
t = float(s) # mais on veut plutôt un nombre, donc on convertit cette chaîne en un flottant
ici
print((t-32)*5/9),"degrés␣celsius")
```

Pour exécuter ce programme, vous pouvez au choix cliquer sur la flèche verte, ou bien utiliser le raccourci clavier `F5`. Si la question vous est posée, choisissez d'exécuter votre programme par l'interpréteur courant, de telle sorte que vous verrez que dans la console il vous est demandé d'introduire une température : vous pouvez essayer 451 ou toute autre température en degrés Fahrenheit bien sûr... ($451^\circ F$ est le nom d'un célèbre roman dystopique de Ray Bradbury)

La température en degrés celsius sera alors affichée, et vous pourrez constater que les variables `s` et `t`, définies dans votre script, ont gardé leurs valeurs dans la console.

Si vous avez besoin d'une autre conversion, vous avez compris qu'il suffit de réexécuter votre script.

Le programme proposé a au moins un écueil : l'affichage du résultat à l'aide de la fonction `print` ne permet guère de réutiliser le résultat du calcul obtenu pour en faire quelque chose. On va modifier notre programme pour faire de l'opération de conversion une fonction, laquelle aura une valeur de retour qu'on pourra nommer, et à partir de laquelle on pourra faire d'autres calculs si on le souhaite.

Modifiez donc votre script pour qu'il ressemble désormais à ceci :

```
def convFahrCelsius(fahr : float) -> float:
    degrésCelsius = (fahr-32)*5/9
    return degrésCelsius
```

puis exécutez-le à nouveau. Outre la ligne `runfile("...")` il ne devrait pas se passer grand chose en apparence, mais en pratique on vient d'introduire une nouvelle fonction qui se nomme `convFahrCelsius`. Il est maintenant possible de l'invoquer pour réaliser autant de conversions que l'on veut, sans avoir à réexécuter notre script à chaque fois. Essayez donc (dans la console) `celsius = convFahrCelsius(451)` par exemple, puis `celsius` ou `print(celsius)`.

Vous aurez noté la syntaxe pour introduire une fonction : une ligne d'entête qui commence par `def`, le nom de la fonction puis un (mais il pourrait y en avoir plusieurs) argument entre parenthèses. On a indiqué ici (c'est optionnel et, à vrai dire, un peu « décoratif », mais cela aide à préciser ce qui est attendu lorsqu'on se relit ou qu'on donne à relire son code) avec des annotations que l'argument attendu est supposé être un nombre flottant (rien n'interdit en pratique qu'il soit un entier par exemple) et que la valeur renvoyée sera également un nombre flottant.

Ensuite le corps de la fonction est en retrait (la coutume est un retrait de 4 caractères) et c'est grâce à ce retrait que l'interpréteur connaît de quoi se compose le corps de la fonction. (D'autres langages utilisent par exemple des caractères tels que {} pour délimiter le corps de la fonction, rien de tel en python)

Enfin, vous aurez noté qu'une variable est définie dans le corps de la fonction. Essayez `print(degrésCelsius)` dans la console. Qu'obtient-on ? On dit de ces variables introduites dans le corps d'une fonction qu'elles sont *locales* à cette fonction. Cela évite de polluer l'espace des noms de nouvelles variables, mais cela permet aussi de réutiliser des noms qui pourraient exister par ailleurs sans risquer d'en écraser la valeur associée.

Modifiez une dernière fois votre script pour qu'il devienne :

```
def convFahrCelsius(fahr):
    celsius = (fahr-32)*5/9
    print(celsius)
```

et exécutez-le encore. Refaites-les mêmes tests que précédemment. Vous aurez noté que la valeur qui s'affiche n'est en fait pas transmise par votre fonction. La moralité est ici que la valeur de retour d'une fonction doit toujours être introduite par la directive `return` et jamais par la fonction `print`.

7 A faire à la maison

D'ici le prochain tp, je vous invite à installer, sur un ordinateur personnel, une distribution de python, sans oublier son environnement de développement Spyder. Deux possibilités : l'une est de se rendre à `https://www.spyder-ide.org`, autrement dit télécharger Spyder qui vient accompagné d'une distribution python, ou d'aller sur `https://www.anaconda.com/download` et télécharger une distribution python qui comprend également l'environnement Spyder. Dans ce dernier cas, il vous faudra sans doute fournir une adresse mail, mais le téléchargement reste, bien entendu, gratuit.

Un autre logiciel qui pourrait vous être bien utile et que je vous recommande chaudement d'installer également : dès cette année pour récupérer les fichiers sources des TP d'informatique, accompagnés de corrections dépliables, et l'an prochain pour les cours et exercices de mathématiques, le logiciel TeXmacs téléchargeable depuis son site `https://www.texmacs.org`