Cours n°1 – 23 septembre 2020

Python :

→ premier contact

→ environnement de travail

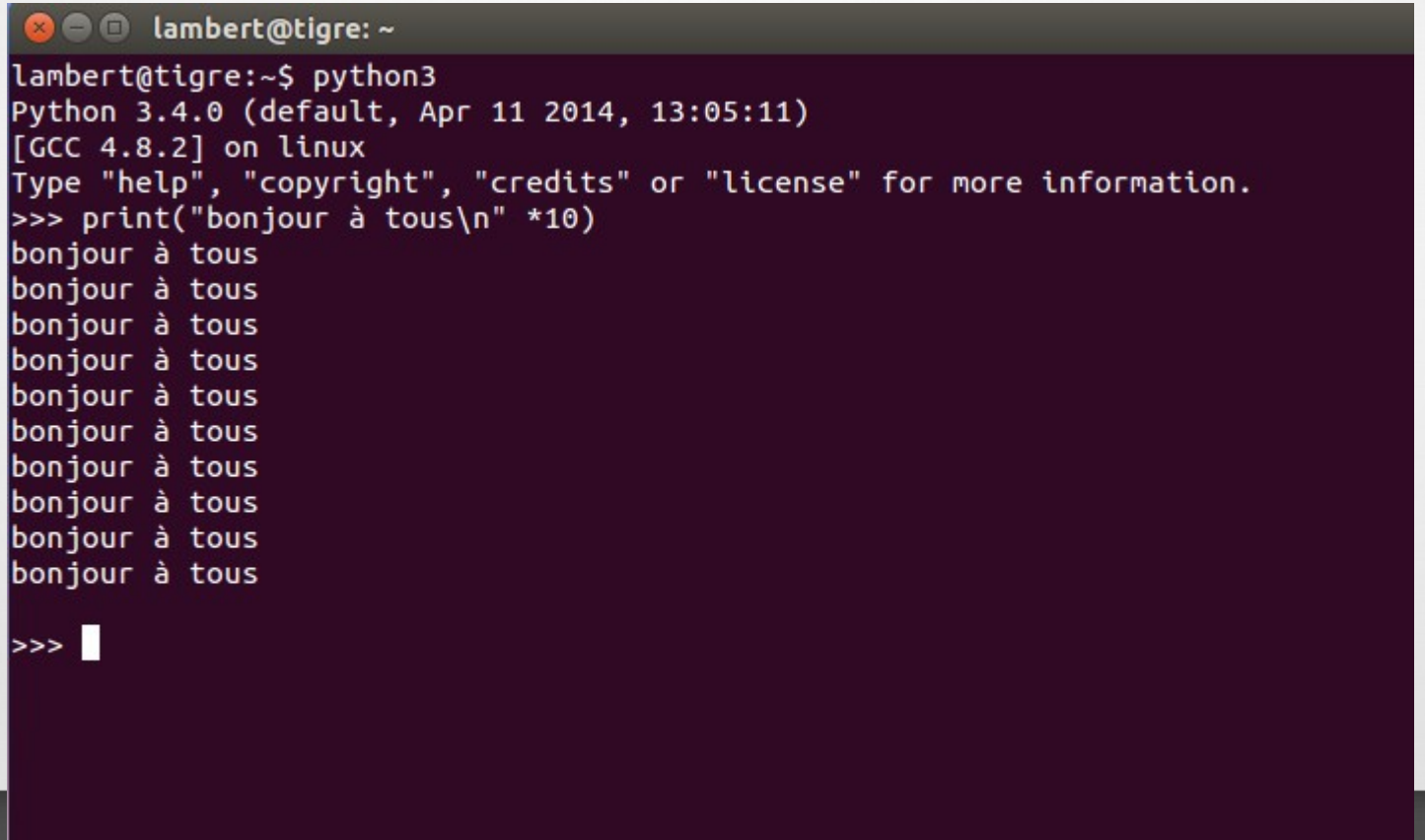
→ variables et affectation,

→ string et tuples

→ conditions (structures de contrôles)

Notre premier écran Python:

- on peut utiliser python directement dans un terminal
- le symbole >>> précise qu'on a la main



Idle

Il est plus pratique et efficace d'utiliser des outils adaptés.

Pendant le cours, nous utiliserons IDLE qui spécialement dédié à Python (et qui se télécharge automatiquement en même temps)

D'autres possibilités peuvent être utilisées , aucune imposée (mais vous devez connaître une façon de vous servir de python!)

Idle

Ce qui permet d'ouvrir des fichiers pour y écrire des scripts variés, de les sauvegarder, puis de les exécuter.

Idle

l'utilisation est très simple:

- on ouvre un fichier (ouverture ou création d'un nouveau fichier avec le menu déroulant)

- on écrit son script

- on le sauvegarde (“CTRL S” ou “enregistrer” ou “enregistrer sous” dans le menu déroulant)

- on l'exécute (F5 ou run dans le menu déroulant)

Idle

remarques:

le caractère # ouvre une ligne de commentaire qui ne sera pas lue par le programme (pas de symbole de fin de commentaire mais un # par ligne, possibilité de commenter/décommenter un paragraphe avec le menu déroulant)

- chaque appel de run remet la mémoire à zéro (affichage de RESTART)
- s'il y a un problème de syntaxe, alors on aura un message d'erreur (voir TP1)

la “calculatrice” Python

On peut utiliser python comme calculatrice.

```
>>> 3+4
7
```

remarques:

- pas besoin de = , mais “enter”
- Les priorités dans les opérations sont “comme en maths” (voir doc2 sur ecampus)
- Par la suite je noterai

  >>>

  quand on est dans l'évaluateur python, pour avoir des repères mais vous n'aurez jamais à écrire >>>

les variables

- règles du choix des noms de variables :(voir doc ecampus)
  - 33 mots réservés interdits
  - pas de caractères spéciaux sauf _(under score) , accents possibles en version3 (pas forcément conseillés)
  - doivent commencer par une lettre

- affectation
  - se fait avec le symbole = (pour une définition ou une modification)

```
>>> x=5
>>> phrase=”bonjour tout le monde”
```

  - le nom est toujours en premier
  - une affectation multiple (plusieurs en même temps) est possible:

```
>>> x, y = 2.4, 8
```

si l'affectation est mal faite, il y aura un message d'erreur

l'affectation

- crée et mémorise un nom de variable
- crée et mémorise une valeur particulière
- crée une association entre le nom et la valeur correspondante rangée dans un “dictionnaire”

```
>>> globals()
{'__builtins__': <module 'builtins' (built-in)>, '__file__': '/usr/bin/idle-python3.2',
    '__package__': None, '__cached__': None, '__name__': '__main__', 'main':
    <function main at 0x7fba7041ae20>, '__doc__': None}
>>> a=3
>>> mot="bonjour"
>>> t=(1,2,3)
>>> c=3+4j
>>> globals()
{'a': 3, 'c': (3+4j), '__builtins__': <module 'builtins' (built-in)>, '__file__':
    '/usr/bin/idle-python3.2', '__package__': None, 't': (1, 2, 3), '__cached__': None,
    '__name__': '__main__', 'mot': 'bonjour', 'main': <function main at
    0x7fba7041ae20>, '__doc__': None}
>>>
```

les types des variables

en Python: pas besoin de déclaration de type pour les valeurs, le typage se fait automatiquement (typage dynamique)

Exemples:

```
>>> x,y,ph=4,15.3,”bonjour”
>>> type(x)
  <class 'int'>
>>> type (y)
  <class 'float'>
>>> type(ph)
  <class 'str'>
>>> type(4)
  <class 'int'>
>>> type((1,3))
  <class 'tuple'>
```

Quels types nous utiliserons?

- Integer : les entiers
- Float: les réels (nombres à virgule)
- String (chaînes de caractères
- T-uples (sortes de n-uplet)
- Listes (ou tableaux)
- Fonctions
- Fichiers

quelques fonctionnalités

x**n ou pow(x,n)

```
>>> 4**3
64
>>> 4**0.5
2.0
```

- % renvoie le reste de la division euclidienne:
  si a=bq+r alors a%b renvoie r

```
>>> 24%5
4
```

- // effectue une division “entière” c'est à dire un quotient

si a=bq+r alors a//b renvoie q

```
>>> 9//2
4
>>> 9/2
4.5
```

Les chaînes de caractères: string

Une chaîne de caractère (string) est une suite de caractères délimitée par des guillemets ou des apostrophes (ces guillemets ou apostrophes ne faisant bien évidemment pas partie de la chaîne). On mettra obligatoirement des guillemets si la chaîne contient une apostrophe et des apostrophes si la chaîne contient des guillemets. On doit impérativement mettre le même symbole au début et à la fin de la chaîne.

On peut additionner 2 chaînes, multiplier une chaîne par un entier, récupérer la longueur de la chaîne (voir TD1)

Attention on ne peut pas additionner une chaîne et un nombre:

```
>>> "bonne année " + 2015
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> "bonne annéee" + str (2015)
'bonne annéee2015'
```

string

Une chaîne est une séquence: les caractères de la chaîne sont numérotés, on dit qu'ils ont un indice. (numéroté à partir de 0).
Si ch est une chaine, ch[i] renverra le caractère de la chaîne dont l'index est i
Il y a une double indexation: de la gauche vers la droite avec des nombres positifs de 0 à n-1 (si n est la taille de la chaîne) et de la droite vers la gauche avec des nombres négatifs commençant à -1.

b	*o*	*n*	*j*	*o*	*u*	*r*		*!*
0	1	2	3	4	5	6	7	8
-9	-8	-7	-6	-5	-4	-3	-2	-1

string

On peut aussi découper des “sous-chaines”, transformer une chaîne en entier ou réel si cela est possible et transformer un nombre en string

!! Attention: on ne peut pas modifier une chaîne

Si mot=”bomjour” on ne peut pas modifier simplement le “m”
L'instruction mot[i]=”n” ne peut pas fonctionner.
mot[i] ne peut servir qu'à récupérer une valeur dans le cas d'une string

Il faudra recopier la chaîne pour la modifier
(donnée non mutable).

Remarque: pas de problèmes avec les accents ou les caractères spéciaux dans la version 3

Les t-uples

Le tuple permet de regrouper des éléments (de n'importe quel type). C'est une donnée “composée” ou composite. C'est une séquence (les éléments sont numérotés).

```
>>> t= (2, 3, 4, 5)

>>> t2=(2,)  # ne pas oublier la virgule

             # pour un tuple à 1 élément

>>> x=()                    #tuple vide

>>>y=(1,(2,3),6)      #tuple contenant un tuple
```

Les t-uples

Comme pour les strings, on a accès aux éléments par leur indice, on peut découper des morceaux mais on ne peut pas modifier un tuple

t=(1,1,2,3,5,8,13,21,34)

1	*1*	*2*	*3*	*5*	*8*	*13*	*21*	*34*
0	1	2	3	4	5	6	7	8
-9	-8	-7	-6	-5	-4	-3	-2	-1

Les t-uples

Autres exemples:

→ des cartes
('valet,'coeur')

(7,'pique')

→ un jeu de carte

(('valet,'coeur'), (7,'pique'), ('dame,'pique'), (10,'pique'))

→ un bon moyen pour transformer un entier de 1 à 12 en nom de mois en français

nomsMois=('janvier','février', 'mars','avril','mai','juin','juillet', 'août', 'septembre','octobre','novembre','décembre')

nomsMois[2] renvoie 'mars'

“Dialoguer” avec la machine : sorties

→ les “sorties”: on utilise print

print permet l'affichage de données (n'importe lesquelles). C'est une fonction donc il y aura des parenthèses.

print n'affiche pas les guillemets ou apostrophes “externes” d'une chaîne de caractères

```
>>> x='bonjour'
>>> x
'bonjour'
>>> print(x)
bonjour
```

“Dialoguer” avec la machine : sorties

→ on peut fabriquer une chaîne regroupant plusieurs données avant de faire l'affichage (voir TD/TP1)
→ mais on peut aussi mettre plusieurs paramètres avec des virgules et utiliser sep= (c'est optionnel)

```
>>> print('bonne','rentrée',2020)
bonne rentrée 2020
>>> print('bonne'+'rentrée'+ str(2020))
bonnerentrée2020
>>> print('bonne'+ " "+'rentrée'+ " "+ str(2020))
bonne rentrée 2020
>>> print('bonne','rentrée',2020, sep="*")
bonne*rentrée*2020
>>> print('bonne','rentrée',2020, sep=" ")
bonne rentrée 2020
>>> print('bonne','rentrée',2020, sep="")
bonnerentrée2020
```

“Dialoguer” avec la machine : entrée

→ les entrées: *il faut pouvoir “saisir” une réponse donnée par l'utilisateur. Pour cela on utilise input*

input (chaine) affiche chaine et attend une entrée clavier; cette fonction renvoit une chaîne de caractères contenant la donnée saisie par l'utilisateur

exemples

On va écrire un petit programme qui affichera ce “dialogue”

bonjour
quel est votre prenom? françoise
comment allez-vous françoise ?
on va faire un calcul
entrez un nombre: 6
entrez un autre nombre: 7
la somme de 6 et 7 est égale à 13

Ce programme va être écrit dans un éditeur pas dans le shell python sinon cela ne peut pas bien fonctionner!

Un premier “script”:

```
print ("bonjour ")
a=input("quel est votre prenom ?  ")
print ("comment allez-vous "+ a+ " ?")
print ("on va faire un calcul")
x=input("entrez un nombre : ")
y=input("entrez un autre nombre : ")
print ("la somme de "+ x+ " et " +  y + " est égale à "+ (x+y))
```

Est-ce juste?

Mais qui est “faux”

bonjour
quel est votre prenom ? françoise
comment allez-vous françoise ?
on va faire un calcul
entrez un nombre : 4
entrez un autre nombre : 1
la somme de 4 et 1 est égale à 41

Le programme a additionné “4” et “1” d'où le 41
→ **il faut transformer les entrées en nombres**

Avec correction :

Il faut modifier la dernière ligne du script:

```
print ("la somme de "+ x+ " et " +  y + " est égale à "+
                str(float(x)+float(y)))
```

```
bonjour
quel est votre prenom ?  jkjl
comment allez-vous jkjl ?
on va faire un calcul
entrez un nombre : 6.8
entrez un autre nombre : 9.3
la somme de 6.8 et 9.3 est égale à 16.1
```

Attention: le str est indispensable si on veut fabriquer une chaîne car “est égale à” + un nombre provoquerait une erreur.

Autre possibilité:

On aurait pu avoir comme dernière ligne du script:

```
print ("la somme de ", x, " et " ,  y, " est égale à ",
 (float(x)+float(y))
```

Dans cette version, pas besoin de str

Remarque : erreur de saisie

Par exemple si faute de frappe:

si l'utilisateur tape 1o à la place de 10
Il y aura une erreur lors de int ou float
On verra plus tard comment tester la chaîne saisie avant de transformer avec int ou float ou comment utiliser le mécanisme d'exception de python

Premières conditions

Très vite dans les programmes on a besoin de “tester” si une réponse est bonne, si une donnée vérifie une propriété.....

test d'égalité:

- pour tester l'égalité de deux “objets” on utilise le test: ==
- Il convient pour **toutes** les données en python
  - l'inégalité se teste avec !=
- Attention: = correspond toujours à une affectation jamais à un test

structure de controle: si..... alors

```
if x==4:
        print ('test vérifié')
```

que se passe-t-il?

- si x n'est pas défini: message d'erreur
- si x vaut 4 : affichage de: test vérifié
- si x ne vaut pas 4 : il ne se passe rien

à remarquer:

- les : en fin de ligne du if qui sont obligatoires
- l'indentation (automatique si éditeur adapté)

si ....alors..... sinon

on peut avoir un “sinon” (jamais obligatoire):

```
if x==2:
        print ('test vérifié')
else:
        print ('test non vérifié')
```

à remarquer:

- les : en fin de ligne
- l'indentation : else doit être aligné avec if
  les deux “actions” (ici print) sont aussi alignées

Un exemple

On veut obtenir le script:

entrez un nombre : 3
entrez un autre nombre : 8
votre réponse pour le calcul de la somme?11
Bravo

entrez un nombre : 7
entrez un autre nombre : 6
votre réponse pour le calcul de la somme?12
ben alors!!! il fallait trouver 13

Et le script:

```
x=float(input("entrez un nombre : "))
y=float(input("entrez un autre nombre : "))
z=float(input("votre réponse pour le calcul de la somme?"))
if z==x+y:
    print ("bravo")
else:
    print ("ben alors!!! il fallait trouver " + str(x+y))
```

On peut mettre plusieurs instructions

```
x= 4
if x==2:
        print ('test vérifié')
        print ('tout va bien')
else:
        print ('test non vérifié')
        print ('il faut recommencer')
        print ('panique à bord')
```

à remarquer:

- les : en fin de ligne
- l'indentation : else doit être aligné avec if
  les deux “blocs” d'actions sont aussi alignés

En python: les blocs d'instructions sont délimités par les niveaux d'indentation.

si ....alors..... sinon si

Le bloc d'instructions pouvant contenir tout ce que l'on veut, on peut y mettre un if :

```
if x==2:
	print ('test vérifié')
else:
	if x== 1:
		print ('valeur particulière')
	else:
		print ('valeur hors norme')
```

elif

elif (qui peut correspondre à “sinon si”) va permettre une écriture plus simple d'un else qui contient un if comme **première** instruction:

```
if x==2:
        print ('test vérifié')
elif  x==1:
        print ('valeur particulière')
else:
        print ('valeur hors norme')
```

attention

Quelle est la différence entre les deux programmes ?

```
if x==2:
        print ('test vérifié')
elif  x==1:
        print ('valeur particulière')
else:
        print ('valeur hors norme')
```

```
if x==2:
        print ('test vérifié')
if x==1:
        print ('valeur particulière')
else:
        print ('valeur hors norme')
```

Attention

Un triangle est déterminé par un tuple des 3 angles du triangle

```
triangle=(45,45,0)
(a,b,c)=triangle  # ou a=triangle[0] etc

if a+b+c!=180:
    print("pas un triangle")
if a==b or a==b or b==c:
    print("isocèle")
```

Là c'est faux!

La bonne version

```
if a+b+c!=180:
    print("pas un triangle")
elif a==b or a==b or b==c:
    print("isocèle")
```

Plusieurs elif

```
if x== 1:
    print ('janvier')
else:
    if x==2:
        print ('fevrier')
    else:
        if x== 3:
            print ('mars')
        else:
```

etc..... c'est un peu lourd!!!

le programme devient:

```
if x== 1:
    print ('janvier')
elif x==2:
    print ('fevrier')
elif x== 3:
    print ('mars')
```

Etc.....

c'est mieux !!! le programme exécute les instructions correspondant à la première condition satisfaite rencontrée (et sort du if ensuite)

Remarque:

Pour cet exemple il est plus rapide d'utiliser un tuple contenant le nom des mois:

```
NomMois=('janvier', 'février', 'mars',
    'avril','mai','juin','juillet','août','septembre','
    octobre','novembre','decembre')

NomMois[i-1]
```

résumé: syntaxe du if

```
if <test1>:
    <bloc instructions1>
elif <test2>:
    <bloc instructions2>
elif <test3>:
    <bloc instructions3>
.....
else:
    <bloc instructions>
```

test supérieur/ inférieur:

<, > , <=, >=
peuvent comparer deux objets *de même type*

```
>>> 4<6
True
>>> "bonjour"<"au revoir"
False
>>> "pomme">"poire"
True
>>> (1,2,3)<(1,2,4)
True
```

test supérieur/ inférieur:

. pour des strings cela correspond à l'ordre alphabétique (attention aux majuscules...)

```
>>> 'poire'<'pomme'
True
>>> 'poire'<'Pomme'
False
```

. Pour des tuples cela correspond à l'ordre lexicographique

. Attention si les objets ne sont pas de même type:

```
>>> 2<"deux"
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    2<"deux"
TypeError: unorderable types: int() < str()
```

le type booléen

Un booléen est un type de données qui ne peut prendre que deux valeurs : vrai ou faux.

En Python, les constantes littérales correspondantes sont notées True et False (attention aux majuscules)

```
>>> False
False
>>>>>> false
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    false
NameError: name 'false' is not defined
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

le type booléen

Tous les types de variables peuvent être interprétés de manière booléenne: une valeur particulière correspond à False et le reste des valeurs à True.

Les valeurs correspondant à False pour les principaux types déjà vus sont:

bool False

int 0

float 0.

string "" ou ''

tuple ()

Un peu de logique

On peut composer des tests avec les connecteurs logiques and, or et not (obligatoirement écrits en minuscules)

```
>>> not(True)
False
>>> not(not(False))
False
```

D'où des tests du style:

```
if not x==2:            # correspond aussi à x!=2
    print ('raté')
if 2<x and x<5:   # pour 2<x<5
   print('compris')
```

and et or

```
if x==2 and y==2:
    print ('test vérifié pour les deux')
elif x== 2 or y== 2:
    print ('test vérifié par l'un des deux et pas les deux')
else:
    print ('aucun des deux ne vérifie le test')
```

Pas de parenthèses obligatoires

(ordre des conditions importants pour éviter des tests inutiles)

Comment cela marche?

- Les conditions sont évaluées dans l'ordre où elles sont écrites
- Pour un or: l'évaluation s'arrête dès qu'une condition n'est pas fausse et renvoie la valeur de cette condition
- Pour un and: l'évaluation s'arrête avec False dès qu'une condition est fausse ou alors avec la valeur de la dernière condition

Exemples:

```
>>> a,b=4,8
>>> a==4 or b==8
True
>>> a<4 or b==8
True
>>> a<4 or b>8
False
>>> a==4 or b<8        # la deuxième condition n'est pas lue
True
>>> a==4 or b8        # donc il ne se produit pas d'erreur ici
True
```

Remarque:

ce qui compte est d'être “faux” ou pas et non d'être la valeur True.

Le script :

```
if a<4 or b:
    print ('OK')
```

va afficher OK

Remarque

si on veux tester que x vaut 3, 4 ou 5 que pensez de

```
>>> x=2
>>> x == 3 or 4 or 5 or 6
4
```

Pourquoi cette réponse?

Car 4 est la première condition vraie rencontrée...

(il teste x==3 qui est faux puis 4 et s'arrête)

Il fallait en fait écrire:

x==3 or x==4 or x==5

(ou faire autrement - voir plus loin)

À retenir

→ utiliser un éditeur adapté
→ les variables, les données
→ indentation “correcte” indispensable
→ input/print
→ if/elif/else
→ and/or