Chapitre 2

Premiers programmes

2.1 Diverses utilisations de `python`

2.1.1 Utilisation en mode interactif

Nous avons vu comment `python` peut s'utiliser en mode interactif dans le shell ou en le lançant dans une console par la commande `python` : cela permet d'exécuter des commandes.

C'est pratique pour utiliser `python` comme une calculatrice ou pour programmer des algorithmes simples. Mais :

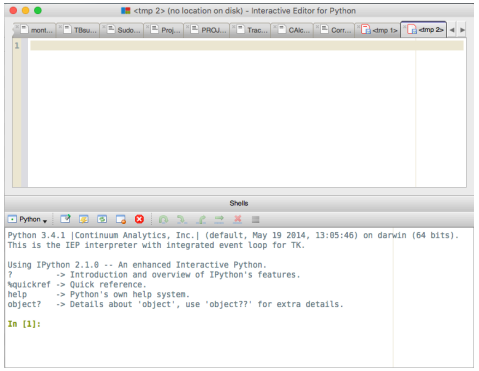
– Le mode interactif ne permet pas de développer des programmes complexes : à chaque utilisation il faut réécrire le programme. La modification d'une ligne oblige à réécrire toutes les autres lignes.

– Pour développer un programme plus complexe on saisit son code (suite d'instructions) dans un fichier texte avant de le faire exécuter par l'interpréteur : plus besoin de tout retaper pour modifier une ligne ; plus besoin de tout réécrire à chaque lancement.

Le programme s'écrit dans un fichier texte que l'on sauvegarde avec l'extension `.py`. Le lancement du programme peut se faire à partir d'une console par la commande `python fichier.py`.

2.1.2 Ecriture d'un programme `python`

Il est préférable d'utiliser un environnement de développement -pyzo par exemple, ou Spyder, ... - :

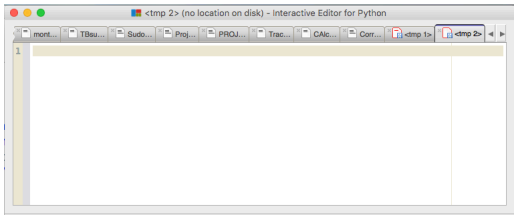


• En haut la fenêtre "éditeur" où l'on tape le code avant de l'interpréter.

• En bas la fenêtre "shell" où l'on saisit des commandes, où l'on interagit avec le programme, et où s'affichent ses résultats.

L'E.D.I. permet d'écrire des programmes, de les sauvegarder, modifier, etc..., de lancer leur exécution, de la stopper, et d'avoir une fenêtre interactive qui aide au développement du programme et retourne les résultats de l'exécution.

• Dans l'environnement de développement la fenêtre éditeur permettra de saisir le programme, de l'exécuter, déboguer avec une meilleure ergonomie.

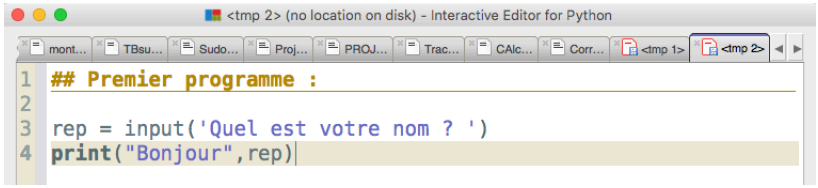


• Les résultats affichés par le programme le seront dans la fenêtre

"shell" (console). L'interaction avec le programme se fera à partir du shell.

2.1.3 Exemple

Exemple : on a écrit dans l'éditeur le petit programme suivant (**script**) :



- La fonction `input` écrit (dans la console) son argument et attend la saisie par l'utilisateur d'une chaîne de caractère, et retourne la valeur saisie. La chaîne de caractère est saisie sans délimiteurs (apostrophes '.' ou guillemets ".").
- Ici la chaine de caractère saisie par l'utilisateur sera affectée à la variable que l'on a appelé `rep`.
- La fonction `print` écrira dans la console ses arguments en les séparant d'un espace. Ici elle a deux arguments.

On exécute le programme en tapant :
– ctrl+e (windows) ou
– cmd+e (MAC) ; apparaît dans le shell :

```
In [1]: (executing lines 1 to 4 of "<tmp 2>")
Quel est votre nom ?
```

On saisit dans le shell notre réponse :

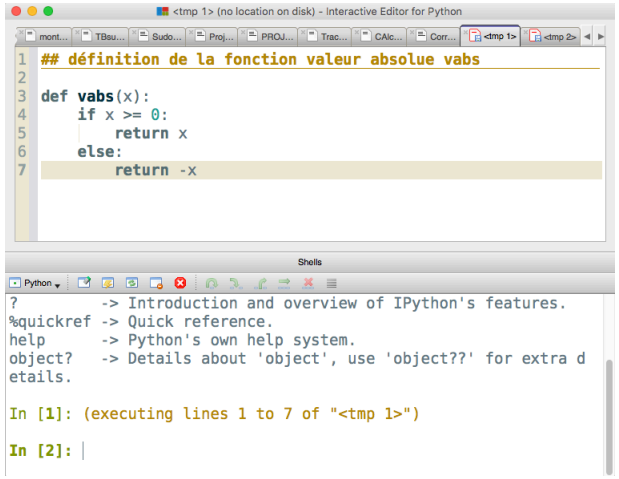
```
In [1]: (executing lines 1 to 4 of "<tmp 2>")
Quel est votre nom ? élève 1/2 de Thiers
```

après avoir appuyé sur entrée l'exécution du programme se poursuit :

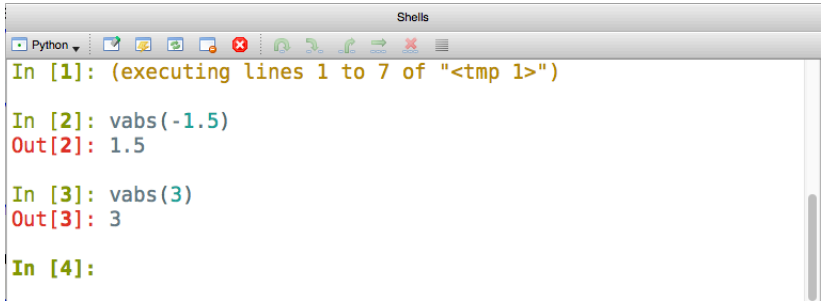
```
In [1]: (executing lines 1 to 4 of "<tmp 2>")
Quel est votre nom ? élève 1/2 de Thiers
Bonjour élève 1/2 de Thiers

In [2]:
```

2.1.4 Autre exemple : définition de fonctions

Si le programme contient des définitions de fonction, leur exécution a pour effet de définir ces fonctions :



Une fois définie la fonction peut être appelée, par exemple dans la console :



La fonction `vabs` retourne la valeur absolue de son argument. Il doit être de type entier ou flottant (autrement produit une erreur).

2.2 Programmation en `python`

2.2.1 Types, expressions

• Les **données numériques** peuvent être de trois types :

- entiers (relatifs), (type `int`),
- nombres à virgule (flottante), (type `float`), et
- nombres complexes (type `complex`).

Type	Nombres	Exemples
`int`	Entiers relatifs	`0, 1, 2, ..., -1, -2, ...`
`float`	Nombres à virgule	`0.1, 1.25, -2.3, 1e10 ...`
`complex`	Nombres complexes	`1j, 1+2j, 2+1.5j, ...`

• On peut leur appliquer les **opérateurs arithmétiques** de base suivants :

`+`	Addition
`-`	Soustraction et signe
`*`	Multiplication
`/`	Division
`**`	Puissance

• les entiers admettent aussi les **opérateurs de division euclidienne** :

`//`	Quotient de la division euclidienne
`%`	Reste de la division euclidienne

• Les données de types **chaines de caractères** (`str`) : On les définit par une suite de caractères entre deux délimiteurs, guillemets `"."` ou apostrophes `'.'` :

```
In [1] : a = "Bonjour l'univers"
In [2] : type(a)
Out[2] : str
```

• **Définition.** Une **expression** est définie par une donnée, une variable définie, ou par des opérations licites appliquées à des variables et données.

Exemples :

- Toute donnée est une expression,
- si `a` est une variable définie de type entier : `a`, `2*a+1`, `(a%2)**a` sont des expressions.

2.2.2 Fonctions d'entrée-sorties

• La fonction `print`.

Elle prend en paramètre une ou plusieurs **expressions** et les affiche dans la console en les séparant d'un espace :

```
In [1] : a = 2
In [2] : print("Le carré de",a,"est",a**2)
Le carré de 2 est 4
```

• La fonction `input`.

Elle attend la saisie par l'utilisateur d'une chaîne de caractère, et retourne la valeur saisie. La chaîne de caractère est saisie sans délimiteurs (apostrophes '.' ou guillemets ".").

On peut passer à la fonction `input` en argument une expression (en général une chaîne de caractère) qui sera inscrite à l'écran.

```
In [1] : ch = input('Saisie : ')
Saisie : ma réponse
In [2] : print('Vous avez saisi :',ch)
Vous avez saisi : ma réponse
```

2.2.3 Type booléen (`bool`) et opérateur de comparaison

• Une donnée de **type booléen** peut prendre 2 valeurs : `True` ou `False`.

Les **Opérateurs de comparaison :**

`a < b`	`a` a une valeur strictement inférieure à celle de `b`
`a <= b`	`a` a une valeur inférieure ou égale à celle de `b`
`a == b`	`a` et `b` ont même valeur.
`a != b`	`a` et `b` ont des valeurs différentes.

et pareillement `a > b` et `a >= b`.
retournent pour valeur un booléen :

```
In [1] : 1 < 2
Out[1]: True
In [2]: 2 <= 1
Out[2]: False
In [3]: type(1<=2)
Out[3]: bool
```

Erreur à éviter :
ne pas confondre l'affectation `=` et la comparaison `==`.

2.2.4 Opérations sur les booléens : connecteurs logiques

On peut effectuer des opérations logiques sur les booléens (par ordre de priorité) : `or`, `and`, `not()` :

`or`	OU logique
`and`	ET logique
`not`	NON logique

```
In [1]: a = (1 <= 2) ; b = (a == False) ; print(a,b)
True False
In [2]: print(a or b, a and b, not(a), not(b))
True False False True
In [3]: var = "chaine"; var == "Chaine"
Out[3]: False
In [4]: print(var != "Chaine", not(var == "Chaine"))
True True
```

```
In [5]: (1 == 1.0) and ("chapeau" == 'chapeau')
Out[5]: True
```

2.2.5 Fonctions de conversion

A tout type de donnée correspond une **fonction de conversion** :

- Elle a même nom que le type considéré,
- elle prend un paramètre, et tente sa conversion dans le type considéré. Erreur lorsque ce n'est pas possible.

```
In [1]: a = 1 ; b = '12'
In [2]: type(a)
Out[2]: int
In [3]: type(b)
Out[3]: str
In [4]: A = str(a) ; B = int(b) ; C = float(b)
In [5]: A
'1'
In [6]: B
12
In [7]: C
12.0
```

• Exemple :

```
In [1]: nbr = input('Saisissez un nombre : ')
Saisissez un nombre : 2
In [2]: print('son carré est',nbr**2)
...
TypeError: unsupported operand type(s) for ** or
pow(): 'str' and 'int'
```

produit une erreur car `nbr` est de type chaine de caractère et ne peut pas être élevé au carré.

Par contre :

```
In [2]: print('son carré est',float(nbr)**2)
son carré est 4.0
```

Pour que l'utilisateur saisisse un nombre, il faut utiliser `input` avec une fonction de conversion.

2.2.6 Conversion en booléen : fonction `bool`

La fonction de conversion `bool` convertit une expression de n'importe quel type en un booléen, selon les règles suivantes :

– Un nombre de type `int`, `float` ou `complex` est converti à `True` s'il est non nul, à `False` sinon.

– Une chaîne de caractère '`str`' est convertie à `True` si elle est non vide, à `False` si c'est la chaîne vide : "".

```
>>> bool(3.14)
True
>>> bool(-7)
True
>>> bool('toto')
True
>>> bool('')
False
```

2.3 Structures de contrôle

2.3.1 Différentes structures de contrôle

`python` est un langage de programmation structuré.
Il reconnaît les structures de contrôles :

– une structure de test SI ... ALORS ... SINON :

`if ... [elif] ... [else]`

– une structure de boucle TANT QUE ...

`while ...`

– une structure de BOUCLE FOR ...

`for ...`

sans lesquelles les programmes s'exécuteraient séquentiellement ligne après ligne.

2.3.2 Structure de test `if ... [else]`

`if` *condition* :

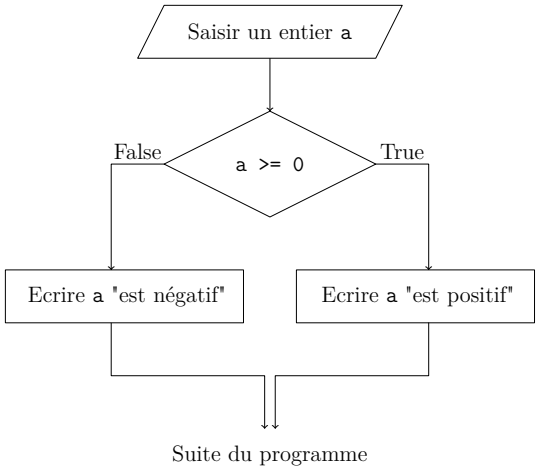
```
.....   Bloc d'Instructions

else
:
.....   Bloc d'instructions
```

meme espace — bloc d'instructions

La condition est évaluée en booléen (en général c'est une expression booléenne, sinon elle est convertit par la fonction `bool`).
– Si elle a valeur `True` le premier bloc d'instruction est exécuté, le deuxième ne l'est pas, puis l'exécution du programme se poursuit.
– Si elle a valeur `False` le deuxième bloc d'instruction est exécuté (le premier ne l'est pas), puis l'exécution du programme se poursuit.
L'instruction `else` est optionnelle.

• **Exemple :**

```
a = float(input('Saisissez un nombre'))
if a >= 0 :
    print(a, 'est positif')
else :
    print(a, 'est négatif')
```



• Exemple : déterminer si un nombre entier est pair ou impair :

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

```
In [1]: pair(3)
Out[1]: False
In [2]: pair(2)
Out[2]: True
```

```
def impair(n):
    return not(pair(n))
```

```
In [3]: impair(3)
Out[3]: True
```

2.3.3 Structure de boucle `for`

• L'instruction `for ... in range(N)` permet de répéter une séquence d'instruction, en boucle, pour une variable qui décrit $0, 1, ..., N-1$.

```
for variable in range(N):
  .....  Instruction1
  .....  Instruction2
  .....  ⋮
  .....  Dernière instruction
```

meme espace — bloc d'instructions

Exemple : calcul (et affichage) de la somme des entiers de 0 à 1000 :

```
S = 0
for k in range(1001):
    S += k
print(S)
```

• L'instruction `for ... in range(n,m)` permet de répéter une séquence d'instruction, en boucle, pour une variable allant de n à $m-1$.

```
for variable in range(n,m):
  .....  Instruction1
  .....  ⋮
  .....  Dernière instruction
```

meme espace — bloc d'instructions

• L'instruction `for ... in range(n,m,k)` permet de répéter une séquence d'instruction, pour une variable allant de n à $m-1$ par pas de k.

```
for variable in range(n,m,k):
  .....  Instruction1
  .....  ⋮
  .....  Dernière instruction
```

meme espace — bloc d'instructions

• **Exemple :** calculer la somme des 1000 premiers termes de la suite de Fibonacci :

```
N = 1000
u, v = 0, 1
S = u+v
for k in range(N-2):
    u, v = v, u+v
    S += v
print(S)
```

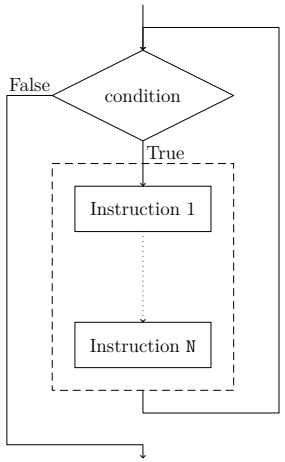
2.3.4 Structure de boucle `while`

• L'instruction `while` *condition* : permet de répéter une séquence d'instruction, tant que *condition* est vérifiée.

```
while condition:
 .....  Instruction1
 .....  ⋮
 .....  Dernière instruction
```

meme espace — bloc d'instructions

En général *condition* est une expression booléenne,
Organigramme dune boucle `while` :



• **Exemple :**

```
x = 1e10
n = 0
while 2**n <= x:
    n += 1
print(n-1 ,"<= log2(" , x , ") <" , n)
```

On obtient à l'exécution :

```
33 <= log2( 1e10 ) < 34
```

Ainsi : $\lfloor \log_2(10^{10}) \rfloor = 33$.

• En général *condition* est une expression booléenne, mais n'importe quelle valeur, de type quelconque peut être utilisée : elle est convertie en booléen.

Exemple : pour attendre la saisie d'une chaîne non vide :

```
# Pour attendre la saisie d'une chaine non-vide
var = ""
while not(var):
    var = input('Saisissez une chaine non-vide : ')
print(var)
```

donne à l'exécution (on aura appuyé deux fois entrée avant de saisir une chaine) :

```
In [1]: (executing lines 1 to 5 of "<tmp 1>")
Saisissez une chaine non-vide :
Saisissez une chaine non-vide :
Saisissez une chaine non-vide : ok
ok
```

2.3.5 Structure de test `if ... [elif] ... [else]`

2.3.6 Exemple : années bissextiles

• Exemple : Dans le calendrier grégorien (actuel), une année est bissextile si :

– Elle est divisible par 4 sans être divisible par 100, ou
– Elle est divisible par 400.

```
def bissextile(n):
    if n%400 == 0:
        return True
    if n%4 == 0:
        if n%100 == 0:
            return False
        else:
            return True
    else:
        return False
```

Remarque : une commande `return` provoque la sortie de la fonction.

• Si dans le dernier programme on remplace les commandes `return` par `print`, le programme bogue :

```
def bissextile(n):
    if n%400 == 0:
        print(n,'est une année bissextile')
    if n%4 == 0:
        if n%100 == 0:
            print(n,"n'est pas une année bissextile")
        else:
            print(n,'est une année bissextile')
    else:
        print(n,"n'est pas une année bissextile")
```

```
>>> bissextile(2000)
2000 est une année bissextile
2000 n'est pas une année bissextile
```

La solution consiste à imbriquer ces deux tests en conditionnant le second à l'échec du premier, de la façon suivante :

```
def bissextile(n):
    if n%400 == 0:
        print(n,'est une année bissextile')
    else:
        if n%4 == 0:
            if n%100 == 0:
                print(n,"n'est pas bissextile")
            else:
                print(n,'est une année bissextile')
        else:
            print(n,"n'est pas une année bissextile")
```

Le programme ne bogue plus :

```
>>> bissextile(2000)
2000 est une année bissextile
```

2.3.7 Structure de test `if [elif] [else]`

L'écriture de tests imbriqués :

```
if (condition1) :
    Bloc d'instructions 1
else :
    if (condition2) :
        Bloc d'instructions 2
    else :
        Bloc d'instructions 3
```

s'écrit à l'aide d'une seule structure de test :

```
if (condition1) :
    Bloc d'instructions 1
elif (condition2) :
    Bloc d'instructions 2
else :
    Bloc d'instructions 3
```

`elif` et `else` sont optionnels. `elif` peut être utilisé plusieurs fois.

• Reprenons l'exemple de la fonction bissextile. En voici une version plus concise et lisible :

```
def bissextile(n):
    if n%400 == 0:
        print(n,'est une année bissextile')
    elif n%4 == 0 and n%100 != 0:
        print(n,'est une année bissextile')
    else:
        print(n,"n'est pas une année bissextile")
```

Le programme ne bogue plus :

```
>>> bissextile(2000)
2000 est une année bissextile
```