Cours 2 : Programmation en Python

Lycée Fénelon

BCPST1

Environnement de développement Python

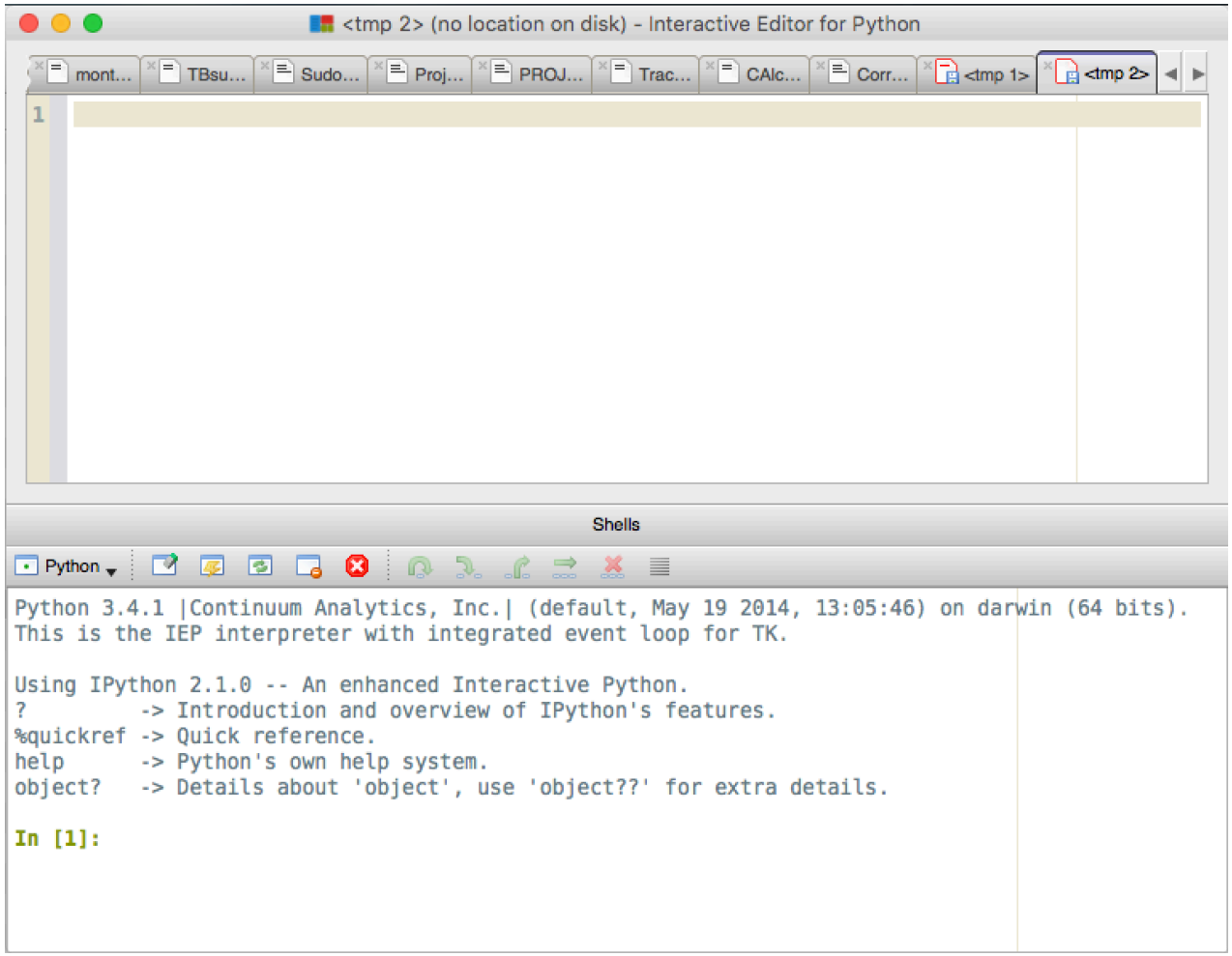
Programmation en python

- Plan
- Types, opérateurs, variables, expressions
- Fonctions
- Structures de contrôle

Ecrire un programme python

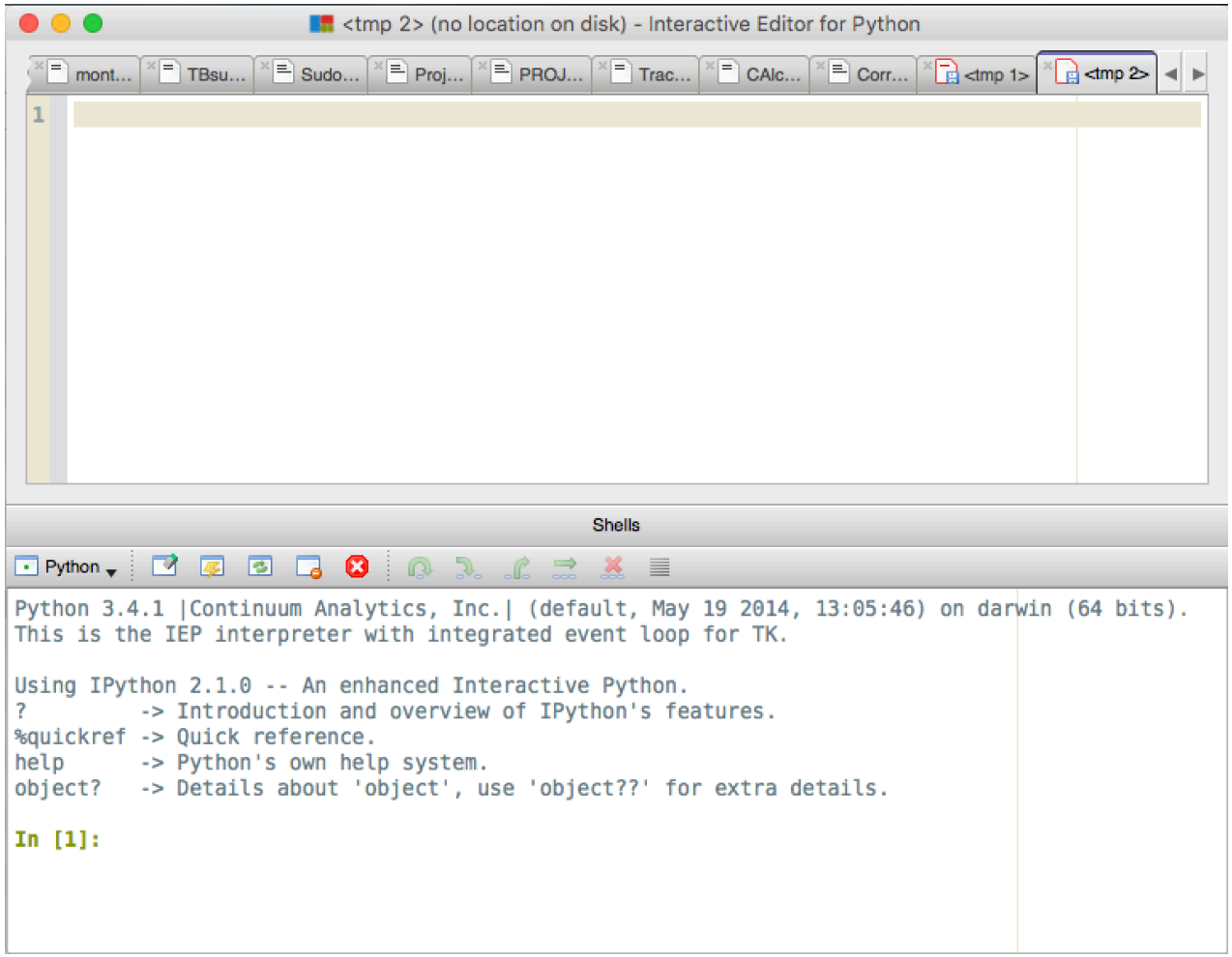
On écrit un programme à l'aide d'un **environnement de développement**, ou E.D.I. ; c'est un logiciel qui permet de saisir, modifier, sauvegarder, exécuter un programme.

Par exemple le logiciel **pyzo** :



Ecrire un programme python

- 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 les résultats.



Bases de programmation

I) Types, opérateurs, variables, expressions

- Données numériques
  - Types de données numériques
  - Opérateurs arithmétiques
- Booléens
  - Opérateurs de comparaison
  - Type booléen
  - Opérateurs logiques
- Séquences : chaînes de caractères et listes
  - Type chaînes de caractères
  - Type liste
  - Opérateurs de concaténation et de duplication
- Variables, expressions
  - Variables
  - Affectation
  - Expressions

Bases de programmation

II) Fonctions

- Fonctions mathématiques
  - Le module `math`
  - Fonctions mathématiques prédéfinies
- Fonctions d'entrée et sortie
  - Fonction de sortie : `print()`
  - Fonction d'entrée : `input()`
- Fonctions de conversion
  - conversion de type : `int()`, `float()`, etc.
- Définition de fonction
  - `def` et `return`
  - Variable locale et variable globale

III) Contrôle du flux d'exécution

- Branchement conditionnel `if ... else`
- Boucle `for`
- Boucle `while`
- Branchement multiple `if ... elif ... else`

Données numériques : types numériques

- 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, ...`

Données numériques : opérateurs arithmétiques

- On peut leur appliquer les opérateurs arithmétiques 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

- Module, partie réelle et imaginaire d'un complexe :
  - La fonction `abs()` renvoie le module de son argument.
  - Si `z` est de type complexe, `z.real` et `z.imag` donnent sa partie réelle et sa partie imaginaire ; ce sont des flottants.

Booléens : opérateurs de comparaison

- Ils permettent de comparer deux valeurs ; ils renvoient un booléen `True` ou `False` :

Opérateurs de comparaison	
==	égalité (de valeur).
!=	non égalité (de valeur).
<	Strictement inférieur (entiers ou flottants).
>	Strictement supérieur (entiers ou flottants).
<=	Inférieur ou égal (entiers ou flottants).
>=	Supérieur ou égal (entiers ou flottants).

```
>>> 1 == 1.0
True
>>> 1 == 2
False
>>> 1 <= 2.0
True
```

Booléens : Type booléen

- Une donnée de type booléen (`bool`) ne peut prendre que deux valeurs : `True` (vrai) et `False` (faux).

Booléens : opérateurs logiques

- Une donnée de type booléen (`bool`) ne peut prendre que deux valeurs : `True` (vrai) et `False` (faux).
- Les opérateurs prenant en opérande des booléens sont les opérateurs logiques : le résultat sera un booléen.

Opérations logiques	
`or`	OU logique.
`and`	ET logique.
`not`	NON logique.

Exemple.

```
>>> a = 1
>>> a < 1e3 and a > 1e-3
True
>>> 1e-3 < a < 1e3
True
```

☞ **Astuce.** python accepte la condition `a < b < c`.

Type chaîne de caractères

- Les données de types chaines de caractères (`str`) : on les définit par une suite de caractères entre deux délimiteurs " . " ou ' . ' :

```
In [1] : "Bonjour l'univers"
```

☞ **Astuce.**
Utiliser les délimiteurs " . " pour déclarer une chaine contenant un apostrophe '.
Utiliser les délimiteurs ' . ' pour déclarer une chaine contenant un apostrophe ".

- Des caractères spéciaux :

\n	saut de ligne
\t	tabulation
\b	décalage d'un caractère à gauche
\"	guillemet
\'	apostrophe

Type liste

Les données de type liste (`list`) : une séquence de données entre crochets, séparés par des virgules :

```
>>> [1, 'ab', 1+1j, 2.0]
```

les données peuvent être de type quelconque.

Séquences : concaténation et duplication

Les données de type liste (`list`) : une séquence de données entre crochets, séparés par des virgules :

```
>>> [1, 'ab', 1+1j, 2.0]
```

les données peuvent être de type quelconque.

- Listes et chaînes de caractères sont tous deux des séquences : ils partagent des opérations et comportements communs :
  - la fonction `len()` :
    - avec en paramètre une liste, renvoie son nombre d'éléments,
    - avec en paramètre une chaîne de caractères, renvoie son nombre de caractères.
  - Opérateurs de concaténation et de duplication :

+	concaténation (en opérandes 2 listes ou 2 chaînes)
*	duplication (en opérandes 1 liste ou 1 chaîne et un entier)

Variables

Une **variable** a :

- un **identifiant** : c'est son nom, qui permet de manipuler la variable. Il peut être constitué de lettres, de chiffres, et du symbôle '_', et ne doit pas débuter par un chiffre. Son nom ne doit pas être un mot-clé réservé du langage.
- Un **contenu**, c'est sa valeur.
- Un **type**, c'est celui de sa valeur.

Variables : déclaration et affectation

Une **variable** a :

- un **identifiant** : c'est son nom, qui permet de manipuler la variable. Il peut être constitué de lettres, de chiffres, et du symbôle '_', et ne doit pas débuter par un chiffre. Son nom ne doit pas être un mot-clé réservé du langage.
- Un **contenu**, c'est sa valeur.
- Un **type**, c'est celui de sa valeur.

• La **déclaration** d'une variable se fait à l'aide d'une affectation :

`variable = valeur`.

• L'opération principale pour une variable est l'**affectation** représentée par le symbôle =. Elle permet de déclarer (c'est à dire définir) une variable, en lui affectant une valeur. Elle permet aussi d'en modifier la valeur.

• Lors d'une affectation l'expression à droite du symbôle = est évaluée, avant d'être affectée à la variable figurant à gauche du symbôle =.

Variables : affectation ; variantes

- L'instruction : `variable †= expression`

est équivalente à l'instruction : `variable = variable † expression`
où † désigne n'importe quelle opération : `+`, `-`, `*`, `/`, `//`, `%`, `**`, `etc.`

- Python permet en une seule instruction d'affectation ('=') d'affecter plusieurs variables. Les valeurs, à droite de '=', sont affectées membre à membre aux variables, à gauche de '='.

```
>>> a, b = 1, 2
>>> a
1
>>> b
2
```

- Bien noter que durant une affectation multiple les valeurs sont celles avant l'appel de l'instruction. Ainsi l'instruction `a,b = b,a` échange les valeurs des 2 variables `a` et `b`.

Expressions

Une expression est tout ce qui est doté d'une valeur.
– Une donnée est une expression.
– Une variable est une expression.
– Si `expr1` et `expr2` sont deux expressions et `expr1` † `expr2` est une opération licite, alors `expr1` † `expr2` est une expression.
– Si `expr` est une expression et † `expr` est une opération licite, alors † `expr` est une expression.
– Si `expr` est une expression, alors `(expr)` est une expression.
– Si `expr1`, ..., `expr.n` sont n expressions, f une fonction prenant n paramètres, alors $f(expr1, ...expr.n)$, si elle est licite, est une expression.

• Exemple : Voici quelques expressions et leur type :
– `1+1j` ; de type complexe,
– `3 * "abc"` ; de type chaîne de caractères,
– `exp(abs(1+1j)**0.5)` ; de type flottant
– `(a == b) or (log(a) <= 1)` ; de type booléen
– `[1, 2] * 3` ; de type liste.

Fonctions mathématiques : le module `math`

- Les fonctions et constantes mathématiques sont à importer à partir du module `math` ; pour importer toute la bibliothèque :

```
from math import *
```

ou pour importer que certaines constantes, fonctions (méthode à privilégier) :

```
from math import pi, cos, sin, tan
```

pour importer la constant `pi` ($\pi \approx 3.1415$) et les fonctions `cos()`, `sin()` et `tan()` ; on peut alors les employer, et ce jusqu'au redémarrage de l'interpréteur :

```
In [1]: cos(pi)
Out[1]: -1.0
```

Fonctions mathématiques du module `math`

- Voici quelques fonctions et constantes utiles du module `math` :

Fonctions et constantes du module `math`	
`pi`	le nombre π
`e`	le nombre $e = \exp(1)$
fonctions mathématiques	
`factorial(n)`	fonction factorielle : $n \mapsto n!$; `n` doit être un entier positif
`floor(x)`	fonction partie entière : $x \mapsto \lfloor x \rfloor$
`sqrt(x)`	fonction racine carrée $x \mapsto \sqrt{x}$; `x` doit être $\geqslant 0$
`exp(x)`	fonction exponentielle : $x \mapsto \exp(x)$
`log(x)`	fonction logarithme népérien : $x \mapsto \ln(x)$; `x` doit être > 0
`log2(x)`	fonction logarithme en base 2 ; `x` doit être > 0.
`log10(x)`	fonction logarithme en base 10.

Fonctions mathématiques du module `math`

- Voici quelques fonctions et constantes utiles du module `math` :

`cos(x)`	fonction cos ; pour `x` exprimé en radians
`sin(x)`	fonction sin ; pour `x` exprimé en radians
`tan(x)`	fonction tan ; pour `x` exprimé en radians
`acos(x)`	fonction arccos ; retourne un angle en radians ; `x` doit être compris entre -1 et 1
`asin(x)`	fonction arcsin ; retourne un angle en radians ; `x` doit être compris entre -1 et 1
`atan(x)`	fonction arctan ; retourne un angle en radians

Fonctions d'entrée et sortie : la fonction print()

- La fonction print() 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 print() admet les deux options :

Option	Type	Effet	Par défaut
sep	str	entre deux arguments	" "
end	str	à la fin	"\n"

Exemple :

```
>>> print("Bonjour","le");print("monde")
Bonjour le
monde
>>> print("Bonjour","le",sep='--',end='**');print("monde")
Bonjour--le**monde
```

Fonctions d'entrée et sortie : la fonction `input()`

- La fonction `input()` attend la saisie par l'utilisateur dans la console d'une chaîne de caractère, et retourne la valeur saisie. La chaîne de caractère est saisie sans délimiteurs (apostrophes '.' ou guillemets ".").
La valeur retournée est une chaîne de caractères.

On peut passer à la fonction `input` en argument une expression (en général une chaîne de caractère) qui sera inscrite dans la console là où est attendue la saisie.

Exemple. Où l'utilisateur saisit : `ma réponse` ;

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

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. Elle produit une 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
```

Fonctions de conversion

- 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.

Fonctions de conversion

- Fonctions de conversion :

Type	Fonction de conversion
entier (`int`)	`int()`
flottant (`float`)	`float()`
complexe (`complex`)	`complex()`
booléen (`bool`)	`bool()`
chaîne de caractères (`str`)	`str()`
liste (`list`)	`list()`

Exemples

```
In [1]: list('abcd')
Out[1]: ['a', 'b', 'c', 'd']

In [2]: str([1,2,3])
Out[2]: '[1, 2, 3]'
```

Fonctions de conversion

- Fonctions de conversion :

Type	Fonction de conversion
entier (`int`)	`int()`
flottant (`float`)	`float()`
complexe (`complex`)	`complex()`
booléen (`bool`)	`bool()`
chaîne de caractères (`str`)	`str()`
liste (`list`)	`list()`

Exemples

```
In [3]: int("3.14")
...
ValueError: invalid literal for int() with base 10: '3.14'

In [4]: int(float("3.14"))
Out[4]: 3
```

Définition de fonction

On peut définir ses propres fonctions. Pour cela la syntaxe est :

```
def nom_de_la_fonction(paramètres) :
    ......   Bloc d'instructions
    ......   return expressions
```

meme espace — bloc d'instructions

- La première ligne (**entête**) commence par `def` suivi du nom de la fonction, et entre parenthèses, de ses paramètres. La ligne finit par deux points `:`.
- Les noms de fonctions suivent les mêmes règles que les noms de variables.
- Le bloc de définition de la fonction est décalé d'un même espace.
- L'instruction `return` *expression* renvoie à celui qui appelle la fonction, le résultat expression.

Définition de fonction

Exemple. Fonction `conjugue()` prenant en paramètre un complexe, et renvoie son conjugué :

```
def conjugue(z): # Renvoie le conjugué de son argument
    x = z.real
    y = z.imag
    return x - y*1j
```

L'interprétation définit la fonction ; on peut alors l'utiliser :

```
In [1]: conjugue(1+2j)
Out[1]: (1-2j)
```

Définition d'une fonction renvoyant le module d'un nombre complexe :

```
def module(z): # Renvoie le module de son argument
    z2 = z*conjugue(z)
    mod = z2.real ** 0.5
    return mod
```

Définition de fonction : renvoyer plusieurs expressions

Une fonction peut renvoyer plusieurs expressions.
Pour cela les séparer d'une virgule :

```
def algebrique(z) # renvoie partie réelle et imaginaire
    reel = z.real
    imag = z.imag
    return reel, imag
```

Pour récupérer les deux expressions utiliser une affectation multiple :

```
In [1]: x, y = algebrique(1+2j)
In [2]: x
Out[2]: 1.0
In [3]: y
Out[3]: 2.0
```

Définition de fonction : paramètre optionnel

Parmi les paramètres d'une fonction, on peut définir une ou plusieurs options.
Les paramètres doivent figurer à la fin, et être munis d'une valeur par défaut :

```
from math import log, e

def logarithme(x, base = e) # log (par défaut népérien)
    return log(x)/log(base)
```

La fonction peut être appelée avec ou sans option passée en paramètre :

```
In [1]: logarithme(1000)
Out[1]: 6.907755278982137

In [2]: logarithme(1000, base = 10)
Out[2]: 2.9999999999999996
```

Définition de fonction : variables locale et globale

Une variable déclarée au sein d'une fonction est dite **locale** : elle est définit à l'appel de la fonction et détruite dès la fin de la fonction.

Une variable déclarée hors d'une fonction est dite **globale** : elle est définit à l'interprétation du script et n'est détruite qu'à l'arrêt de l'interpréteur.

On a exécuté le script suivant :

```
x = 1          # variable globale

def f():
    x = 2      # variable locale
    y = 3      # variable locale
    print(x,y)
```

Définition de fonction : variables locale et globale

On a exécuté le script suivant :

```
x = 1          # variable globale

def f():
    x = 2      # variable locale
    y = 3      # variable locale
    print(x,y)
```

puis :

```
In [1]: f()
2 3

In [2]: x
Out[2]: 1

In [3]: y
... NameError : name y is not defined
```

Structures de contrôle

python est un langage de programmation structuré.

Il reconnaît les structures de contrôles :

– un branchement conditionnel SI ... ALORS ... SINON :

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

– un branchement multiple SI ... ALORS ... SINON SI ... SINON :

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

– une structure de boucle TANT QUE ...

```
while ...
```

– une structure de BOUCLE FOR ...

```
for ...
```

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

Branchement conditionnel `if ... else`

`if` *condition* :

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

`else` :

`.....` `Bloc d'instructions`

meme espace — bloc d'instructions

`condition` est une expression booléenne.

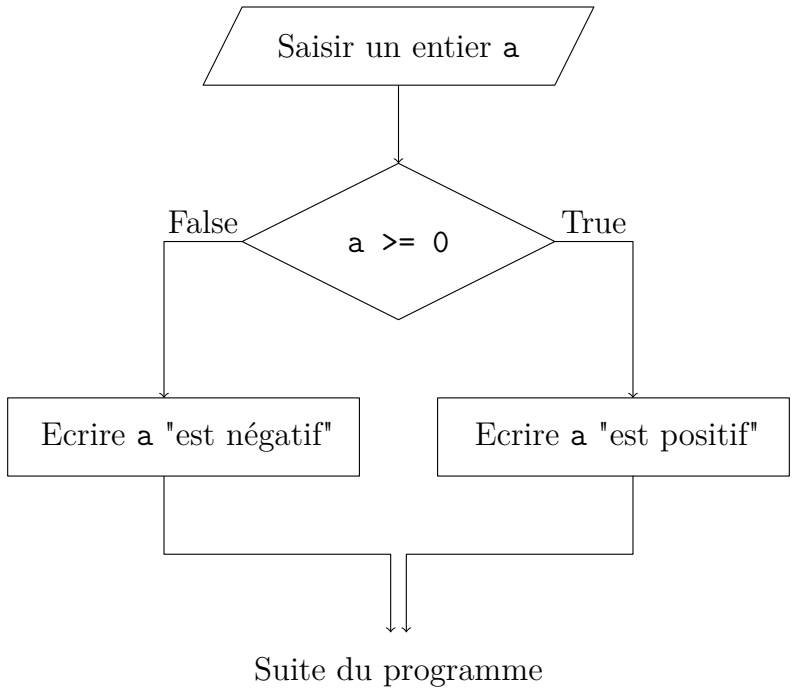
– Si elle a valeur `True` le premier bloc d'instruction est exécuté, le deuxième ne l'est pas, puis l'exécution du programme se poursuit.

– 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.

Branchement conditionnel if ... else

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



Branchement conditionnel `if ... else`

- 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
```

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)
```

Boucle for

- 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

Boucle for

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)
```

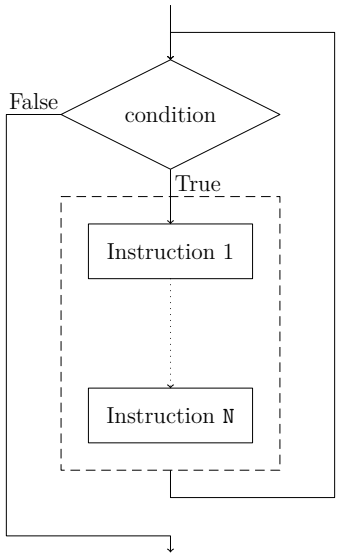
Boucle `while`

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

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

En général *condition* est une expression booléenne,

Organigramme d'une boucle `while` :



Boucle while

Exemple : algorithme d'Euclide pour le calcul du plus grand commun diviseur de 2 nombres :

```
def pgcd(a,b):
    while b != 0:
        a, b = b, a%b
    return a
```

Exemple :

```
In [1]: pgcd(12,18)
Out[1]: 6
```

Le plus grand diviseur commun à 12 et 18 est 6.

Sortie anticipée d'une boucle : break

- Au sein d'une boucle (for ou while), l'instruction break provoque la sortie de la boucle ; le programme se poursuit après la boucle.
- C'est très pratique pour écrire plus simplement certaines boucles.

Exemple 1 : Demander à l'utilisateur de répondre par oui ou par non, tant que la réponse donnée n'est ni oui ni non.

Méthode 1 :

```
reponse = ""
while reponse != "oui" and reponse != "non":
    reponse = input("Répondre par 'oui' ou 'non' ")
```

Méthode 2 : avec l'instruction break :

```
while True:    # Boucle perpétuelle
    reponse = input("Répondre par 'oui' ou 'non' ")
    if reponse == "oui" or reponse == "non":
        break    # Fin de boucle
```

Sortie anticipée d'une boucle : break

- Au sein d'une boucle (for ou while), l'instruction break provoque la sortie de la boucle ; le programme se poursuit après la boucle.
- C'est très pratique pour écrire plus simplement certaines boucles.

Exemple 2 : L'utilisateur dispose de 6 possibilités pour découvrir un nombre mystère.

Méthode 1 : Avec une boucle while

```
mystere = randint(1,100)    # Entier 1 ⩽ . ⩽ 100 tiré au sort
tour = 0
while tour < 6:
    reponse = input("Quel est le nombre mystère ? ")
    if int(reponse) == mystere:
        print("gagné !")
        tour = 6
    else:
        tour += 1
```

Sortie anticipée d'une boucle : break

- Au sein d'une boucle (for ou while), l'instruction break provoque la sortie de la boucle ; le programme se poursuit après la boucle.
- C'est très pratique pour écrire plus simplement certaines boucles.

Exemple 2 : L'utilisateur dispose de 6 possibilités pour découvrir un nombre mystère.
Méthode 2 : Avec une boucle for et l'instruction break

```
mystere = randint(1,100)     # Entier 1 ⩽ . ⩽ 100 tiré au sort
for tour in range(6):
    reponse = input("Quel est le nombre mystère ? ")
    if int(reponse) == mystere:
        print("gagné !")
        break    # Sortie anticipée
```

Branchement multiple 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.

Branchement multiple if ... elif ... else

• Exemple : fonction prenant en paramètre les coefficients réels a, b, c d'un trinôme $ax^2 + bx + c$, et qui affiche dans la console ses racines :

```
def trinome(a,b,c): # donne les racines de ax^2+bx+c
    D = b**2-4*a*c # Discriminant
    if D > 0:      # Discriminant > 0
        print("2 racines réelles distinctes")
        d = D**0.5
        print((-b-d)/(2*a), (-b+d)/(2*a))
    elif D < 0:    # Discriminant < 0:
        print("2 racines complexes conjuguées")
        d = (-D)**0.5
        print((-b-1j*d)/(2*a), (-b+1j*d)/(2*a))
    else:          # Disciminant = 0:
        print("1 racine réelle")
        print(-b/(2*a))
```

Branchement multiple `if ... elif ... else`

Exemples :

avec $x^2 - 2x + 1$:

```
In [1]: trinome(1,-2,1)
1 racine réelle
1.0
```

avec $x^2 - 4$:

```
In [2]: trinome(1,0,-4)
2 racines réelles distinctes
-2.0 2.0
```

avec $x^2 + 4$:

```
In [3]: trinome(1,0,4)
2 racines complexes conjuguées
-2j 2j
```

Branchement multiple if ... elif ... else

Autre exemple : module et argument d'un nombre complexe.

```
from math import acos

def formeExponentielle(z):
    Z2 = z*conjugue(z)        # |z|^2 = z z̄
    module = Z2.real ** 0.5 # |z| = √(z z̄)
    if module == 0:           # Cas |z| = 0
        return module
    elif z.imag >= 0:         # Cas arg(z) ∈ [0, π]
        z1 = z/module
        argument = acos(z1.real)
    else:                     # Cas arg(z) ∈ ]-π, 0[
        z1 = z/module
        argument = -acos(z1.real)
    return module, argument
```

Branchement multiple `if ... elif ... else`

Exemples :

$$1 + i = \sqrt{2} \cdot e^{i\frac{\pi}{4}}$$

```
In [1]: formeExponentielle(1+1j)
Out[1]: (1.4142135623730951, 0.7853981633974484)
```

```
In [2]: formeExponentielle(0)
Out[2]: 0.0
```

$$-i = e^{-i\frac{\pi}{2}}$$

```
In [3]: formeExponentielle(-1j)
Out[3]: (1.0, -1.5707963267948966)
```