Syntaxe Python

ITC

Types de base

Les objets Python possèdent tous un type qui est déterminé à la volée lors de l'exécution du code.

	création	*opérations*	*remarques*
Entiers (**int**)	`0`, `12`, `-4`	`+`, `-`, `*`, `//`, `%`, `**`	Précision illimitée
Flottants (**float**)	`12.2`, `2.`, `-4.2`, `1.4e-12`	`+`, `-`, `*`, `/`, `**`	Précision à 10^{-16} près
Booléens (**bool**)	`True`, `False`	**and**, **or**, **not**	

Comparaisons

Les comparaisons sont des opérations qui renvoient un booléen :

`==`, `!=`, `<`, `>`, `<=`, `>=`

Variables et Fonctions

Une variable est un *nom* associé à un objet.

	syntaxe	*description*
Affectation	`nadine = ...`	associe le nom `nadine` à l'objet …
Lecture	`nadine`	renvoie l'objet associé à `nadine`
Mise à jour	`x += n`, `x -= n`, …	équivaut à `x = x + n`, `x = x - n`, …

Définition d'une fonction :

```
def f(a,b,...) :
    # Block d'instructions
    return ...
```

Appel d'une fonction :

```
f(12,True,...)
```

Listes

Les *listes* sont des séquences d'objets de taille *variable*

	syntaxe	*remarques*
Création	`['N',12,True,21.0]`, `[]`	Une liste peut être vide
Compréhension	`[ e for x in s ]`	`e` est une expression, `s` est une séquence
Longueur	`len(L)`	
Accès	`L[i]`	`i` est un entier, `0 ⩽ i < len(L)`
Concaténation	`L1 + L2`	expression qui renvoi une liste
Répétition	`L * n`	`L + L + ... + L`, n fois
Tranches	`L[i:j]`, `L[:]`, `L[i:j:k]`	`L[:j] (i=0)`, `L[i:] (j=len(L))`
Copie	`L.copy()`	Réalise une copie *superficielle*

	syntaxe	*remarques*
Ajout	`L.append(x)`	Ajoute `x` à la fin de `L`
Dépiler	`x = L.pop()`	Enlève le dernier éléments de `L` et le renvoie

Tuples

Les *tuples* sont des séquences d'objets de taille *fixe*

	syntaxe	*remarques*
Création	`('N',12,True,21.0)`, `()`, `(1,)`	Un tuple peut être vide
Longueur	`len(T)`	
Accès	`T[i]`	`i` est un entier, `0 ⩽ i < len(L)`
Concaténation	`T1 + T2`	**Attention** ce n'est pas l'addition…
Répétition	`T * n`	`T + T + ... + T`, n fois
Tranches	`T[i:j]`, `T[:]`, `T[i:j:k]`	`T[:j] (i=0)`, `T[i:] (j=len(T))`
Dépaquetage	`x0, ... ,xn = T`	Avec `T` un tuple de $n+1$ éléments

Le dépaquetage permet d'écrire : `a, b = c, d` ce qui équivaut à `a, b = (c,d)`. En particulier on peut écrire `a, b = b, a`.

De la même manière, `return a,b, ... ,c` est équivalent à `return (a,b, ... ,c)`.

Chaînes

Les *chaînes* sont des séquences de caractères de taille *fixe*

	syntaxe	*remarques*
Création	`"Nadine"`, `'Maline'`, `""`, `''`	Une chaîne peut être vide.
Longueur	`len(S)`	
Accès	`S[i]`	`i` est un entier, `0 ⩽ i < len(L)`
Concaténation	`S1 + S2`	
Répétition	`S * n`	`S + S + ... + S`, n fois
Tranches	`S[i:j]`, `S[:]`, `S[i:j:k]`	`S[:j] (i=0)`, `S[i:] (j=len(S))`

Quelques caractères particuliers et fonctions importantes sur les caractères.

`'\n'`	retour à la ligne
`'\t'`	tabulation
`'\\'`	backslash
`"\""`	" entre deux "
`'\''`	' entre deux '
`ord``(c)`	code ASCII du caractère `c`
`chr``(n)`	caractère dont le code est `n`

Le générateur **range**

`range``(a,b,p)` est un *générateur*. Ce **n'est pas** une liste. Il génère les entiers entre `a` (inclus) et `b` (**exclus**) par pas de `p`. On peut omettre `p`.

	syntaxe	*remarques*
Création	**`range`**`(a,b)`	`a` est inclus, `b` est exclus
Création avec pas	**`range`**`(a,b,p)`	Si `p = -1`, ordre décroissant

Parcours

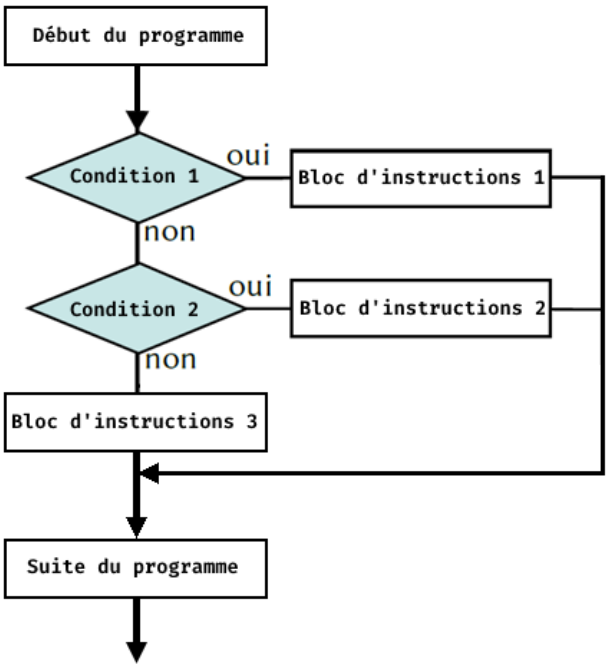
Une séquence est une liste, un générateur, un tuple ou un chaîne. Une séquence peut être parcourue par une boucle **`for`**.

```
for x in S :
    # Block d'instructions
# Suite du programme
```

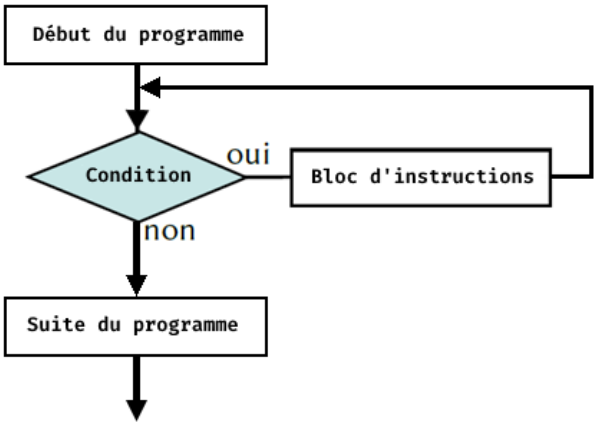
Notez que le parcours d'un dictionnaire est un parcours du générateur des *clés* du dictionnaires (**`for`** `k` **`in`** `D.keys()`) ou des associations (**`for`** `k,v` **`in`** `D.items()`).

Structures de contrôle

```
if condition1 :
    # Block d'instructions 1
elif condition2 :
    # Block d'instructions 2
else :
    # Block d'instructions 3
#Suite du programme
```




```
while condition :
    # Block d'instructions
    ...
#Suite du programme
```



L'instruction **break**, *si elle est exécutée* met fin à l'exécution de la boucle, à la profondeur maximale, qui contient l'instruction.

```
for x in S :
    # Block d'instructions
    break # Fin du parcours
    # Block d'instructions
# Suite du programme
```

```
while condition :
    # Block d'instructions
    break # Fin de la boucle
    # Block d'instructions
# Suite du programme
```