

Syntaxe Python

PSI/PSI*

Types de base

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

	<i>création</i>	<i>opérations</i>	<i>remarques</i>
Entiers (int)	0, 12, -4	+, -, *, //, %, **	Précision illimitée
Floattants (float)	12.2, 2., -4, 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.

	<i>syntaxe</i>	<i>description</i>
Affectation	nadine = ...	associe le nom nadine à l'objet ...
Lecture	nadine	renvoie l'objet associé à nadine

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*

	<i>syntaxe</i>	<i>remarques</i>
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, $0 \leq i < \text{len}(L)$
Concaténation	L1 + L2	expression qui renvoie une liste
Concaténation	L1 += L2	effet de bord, modifie L1
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))

	<i>syntaxe</i>	<i>remarques</i>
<i>Ajout</i>	<code>L.append(x)</code>	Ajoute x à la fin de L
<i>Dépiler</i>	<code>x = L.pop()</code>	Enlève le dernier éléments de L et le renvoie

Tuples

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

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

Chaînes

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

	<i>syntaxe</i>	<i>remarques</i>
<i>Création</i>	<code>"Nadine", 'Maline', "", ''</code>	Une chaîne peut être vide.
<i>Longueur</i>	<code>len(S)</code>	
<i>Accès</i>	<code>S[i]</code>	i est un entier, $0 \leq i < \text{len}(L)$
<i>Concaténation</i>	<code>S1 + S2</code>	
<i>Répétition</i>	<code>S * n</code>	<code>S + S + ... + S</code> , n fois
<i>Tranches</i>	<code>S[i:j], S[:], S[i:j:k]</code>	<code>S[:j]</code> (i=0), <code>S[i:]</code> (j=len(S))
<i>Conversion</i>	<code>list(S)</code>	

Quelques caractères particuliers et fonctions importantes sur les caractères.

<code>'\n'</code>	retour à la ligne
<code>'\t'</code>	tabulation
<code>'\\'</code>	backslash
<code>'\"'</code>	" entre deux "
<code>'\''</code>	' entre deux '
<code>ord(c)</code>	code ASCII du caractère c
<code>chr(n)</code>	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.

	<i>syntaxe</i>	<i>remarques</i>
<i>Création</i>	<code>range(a,b)</code>	a est inclus, b est exclus
<i>Création avec pas</i>	<code>range(a,b,p)</code>	Si p = -1, ordre décroissant
<i>Conversion</i>	<code>list(range(a,b,p))</code>	

Dictionnaires

Les *dictionnaires* permettent d'enregistrer un nombre variable d'associations *clé, valeur*. Les clés peuvent être de type entier, flottant, booléen, chaîne ou tuple. Les valeurs peuvent être un objet quelconque.

	<i>syntaxe</i>	<i>remarques</i>
<i>Création</i>	<code>{'N' : 12, True : 21.0}, {}</code>	Un dictionnaire peut être vide
<i>Longueur</i>	<code>len(D)</code>	
<i>Accès</i>	<code>D[k]</code>	k est une clé
<i>Présence</i>	<code>k in D</code>	
<i>Ajout</i>	<code>D[k] = e</code>	
<i>Clés</i>	<code>D.keys()</code>	générateur des clés
<i>Associations</i>	<code>D.items()</code>	générateur des associations

Parcours

Une séquence est une liste, un dictionnaire, 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 des *clés* du dictionnaires (x est un clé de S). Les parcours suivants sont équivalents :

```

for k in D :
    # Block d'instructions
# Suite du programme

```

```

for k in D.keys() :
    # Block d'instructions
# Suite du programme

```

```

for k in D :
    v = D[k]
    # Block d'instructions
# Suite du programme

```

```

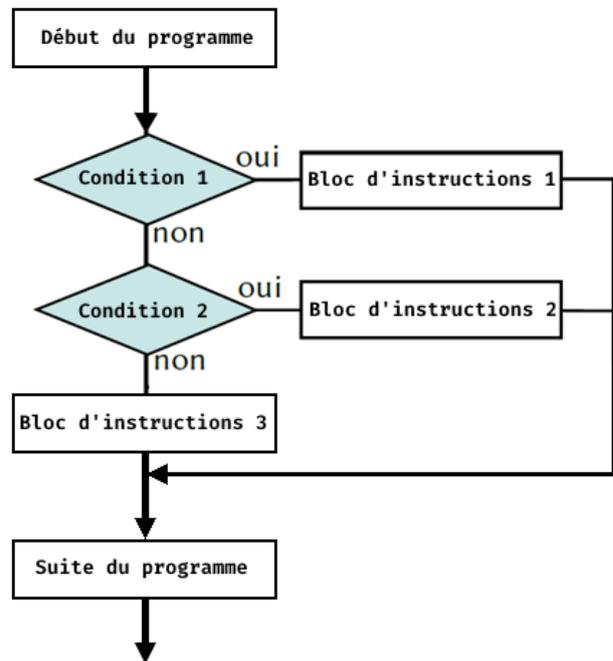
for k, v in D.items() :
    # Block d'instructions
# Suite du programme

```

Structures de contrôle

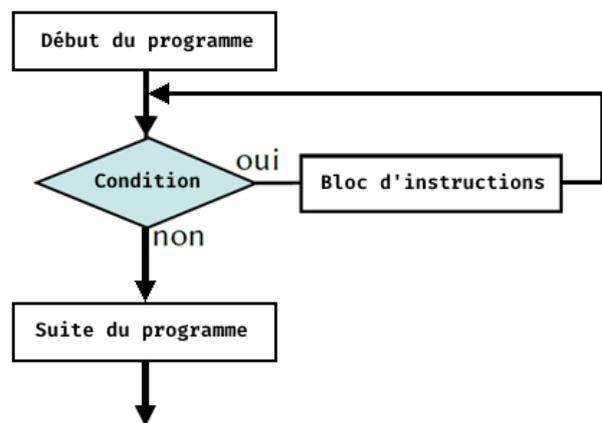
```

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
    
```

Complexités

Les complexités devraient être rappelées en début de sujet. Toutefois il est bon de connaître les complexités des éléments de la fiche de syntaxe.

Toutes les opérations de la fiche de syntaxe qui ne sont pas ci-dessous sont en $O(1)$

La complexité d'une liste en compréhension est la même complexité que celle de la boucle **for** équivalente (c.f. cours).

<i>instructions</i>	<i>complexité</i>	<i>remarques</i>
<code>x ** n</code>	$O(n)$	
<code>S1 = S2</code>	$O(\min(\text{len}(S1) + \text{len}(S2)))$	S1 et S2 sont des listes, chaînes ou tuples
<code>S1 != S2</code>	$O(\min(\text{len}(S1) + \text{len}(S2)))$	S1 et S2 sont des listes, chaînes ou tuples
<code>S1 + S2</code>	$O(\text{len}(S1) + \text{len}(S2))$	S1 et S2 sont des listes, chaînes ou tuples
<code>L1 += L2</code>	$O(\text{len}(L2))$	L1 et L2 sont des listes
<code>S * n</code>	$O(\text{len}(S1)*n)$	S est une liste, une chaîne ou un tuple
<code>S[i:j:p]</code>	$O((j-i)//p)$	S est une liste, une chaîne ou un tuple
<code>list(S)</code>	$O(\text{len}(S))$	S est range , une chaîne ou un tuple

•••

La complexité d'une conditionnelle (**if**) est la plus grande des complexités des blocs d'instructions : $O(\max(C_1, C_2, \dots, C_n))$ pour n branchements.

•••

La complexité d'un parcours (**for**) est la somme des complexités du block d'instruction à chaque itération : $O(\sum_{x \in S} C_x)$ pour un parcours de S.

•••

La complexité d'une boucle while est la somme des complexités du block d'instruction à chaque itération et de la complexité de la vérification de la condition à chaque entrée de la boucle : $O(C_n + \sum_{i=0}^{n-1} B_i + C_i)$ pour n itérations, avec B_i la complexité du block, C_i la complexité de la vérification de la condition, C_n la vérification de la condition juste avant la fin de boucle.

•••

Divers

	<i>syntaxe</i>	<i>remarques</i>
<i>Affichage console</i>	<code>print(o)</code>	o est un objet
<i>Commentaire</i>	<code># Nadine</code>	
<i>Modules</i>	<code>import m</code>	m.objet est disponible
<i>Alias de Modules</i>	<code>import module as m</code>	m.objet est disponible
<i>Nom dans un module</i>	<code>from m import objet</code>	objet est disponible
<i>Fichiers</i>	<code>open, write, read, realine, readlines, split, close</code>	la documentation vous sera donnée
<i>Assertion</i>	<code>assert condition</code>	