Labo d'introduction à l'informatique

pour les mathématiques

Yann Thorimbert

UNIVERSITÉ DE GENÈVE
CENTRE UNIVERSITAIRE D'INFORMATIQUE

Semaine 7.2

Slicing de listes

Chaînes de caractères

Chaînes de caractères

- "Chaîne" → ***string*** en anglais.

- Réutilise le concept de **tableau** (ou liste, cf. différence vue en cours) pour représenter des suites ordonnées de caractères, au lieu de suite ordonnées de nombres.

- Rappel : ASCII (par exemple) établit une **correspondance** nombre ↔ caractères. Un tableau de nombres peut donc être interprété comme un tableau de caractères.

Correspondance avec code ASCII

```
n = ord("A")
print(n) # affiche 65


c = chr(65)
print(c) # affiche 'A'
```

Déclaration d’une chaîne en Python

- Les **guillemets** indiquent à Python que l'on déclare une suite de caractères :
```
texte = "Salut !"
# plus pratique que texte = [83,97,108,117,116,32,33]
```

- L'**apostrophe** tient le même rôle :
```
texte = 'Salut !'
```

- L'un peut être utilisé dans l'autre :
```
texte = 'Bob est mon "ami", en quelques sortes'
```

  (cf. test Moodle pour plus de détails)

Type d'une chaîne

- Une chaîne de caractères définie comme précédemment est un objet Python de type `str`.

  ```
  x = "Hello"
  ```

  Question : quel est le **type** Python de la variable x ci-dessus ?
  Réponse : x est de type "str".

- Tolérance zéro dorénavant durant les questions orales avec les questions sur les types `int`, `float`, `bool` et `str`.

- Il est impossible de comprendre les messages d'erreurs sans connaître le nom des types que vous utilisez.

Longueur d'une chaîne

- `len s’utilise comme pour les listes.`
- Tous les caractères sont comptés (espaces, ponctuation, …).

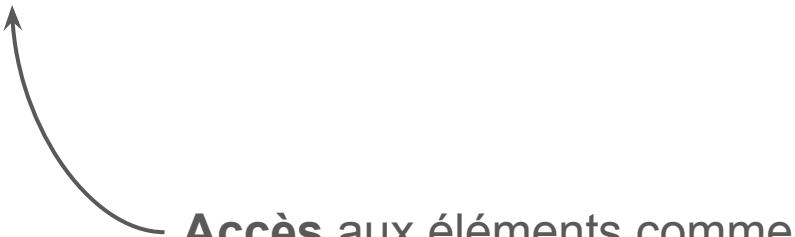
```
texte = "Salut !"
n = len(texte)
print(n) # affiche 7
```

Exemple de parcours d’un texte

Affichage des lettres une par une :

```
texte = "Salut !"

for c in texte:
    print(c)
```



Accès aux éléments comme pour une liste !

Exemple de parcours d’un texte

On accède aux éléments individuels avec des crochets, toujours comme pour les listes.

```
texte = "Salut !"

for i in range(len(texte)):
    print(c[i])
```

Comparaison de caractères

On peut utiliser l'opérateur == pour comparer un ou plusieurs caractères.

```
texte = "Salut !"

for i in range(len(texte)):
    if c[i] == "u":
        print(f'Il y a un "u" en position {i}.')
```

Concaténation de chaînes

L'opérateur + est défini comme concaténateur de ses arguments :

```
a = "Salut"
b = "les amis"

c = a + " " + b
print(c) # affiche "Salut les amis"
```

Répétition de chaînes

L'opérateur * est défini entre un str et un int comme produisant une répétition de la chaîne :

```
a = "Salut"
print(a*3) # affiche "SalutSalutSalut"
```

Opérateurs non-définis

Notez bien que les opérateurs `/` et `-`, par exemple, ne sont pas du tout définis pour les strings. Les personnes ayant créé Python auraient pu ne pas faire le choix de définir * pour les strings, comme c'est le cas de plein de langages.

Observez l'effet de la ligne suivante dans votre console :

```
x = "Salut" -  "tout le monde"
```

Il est capital de comprendre la signification des messages d'erreurs !

```
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Signifie "erreur de type : l'opérateur - n'est pas défini entre un `str` et un `str`"

Immuabilité des chaînes en Python

En Python, on ne peut modifier une chaîne déjà créée.

Exemple de code produisant une erreur :

```
x = "Salut moi !" #ici, x[6] vaut "m"
x[6] = "t"
```

```
TypeError: 'str' object does not support item assignment
```

Immuabilité des chaînes en Python

En Python, on ne peut modifier une chaîne déjà créée, il faut en créer une nouvelle. Cela n'empêche pas de réutiliser une chaîne précédente dans l'affectation de la valeur à une nouvelle chaîne du même nom :

```
texte = "Salut"
texte = texte + " tout le monde"
print(texte) # "Salut tout le monde"
texte += " !"
print(texte) # "Salut tout le monde !"
```

Slicing

Le slicing permet de sélectionner une sous-partie d'une séquence.

```
x = "Salut moi !" #ici, x[4] vaut "t"
print(x[0:5]) # "Salut"
```

Signifie : "obtenir la sous-séquence de x de l'élément numéro 0 jusqu'au 5 non-compris"

Slicing

- Syntaxe générale similaire à celle des arguments de `range`.
- `x[a:b:step]` produit une sous séquence de l'élément d'indice a jusqu'à b non compris, avec des "sauts" de `step`.
- On peut omettre a, b et `step` autant que l'on veut ! Dans l'exemple ci-dessous, on omet l'indice de fin et l'incrément.

```
x = "Salut moi !" #ici, x[6] vaut "m"
y = x[0:6] + "t" + x[7:]
print(y) # "Salut toi !"
```

Slicing

- Le slicing peut être utilisé sur les listes également :

```
x = [2, 5, 7, 1, 3]
print(x[::2]) # [2, 7, 3]
```

- Tous les effets du slicing peuvent être produits avec une boucle.

- En Python, on préfère souvent la concision du code et les slices sont très utilisés.

- Enfin, Python appelle en coulisse du code C optimisé pour ce genre de cas de figure ⇒ souvent plus performant d'utiliser le slicing qu'écrire sa propre boucle pour obtenir le même effet.

Méthodes des objets de type `str`

- Nous verrons très prochainement qu'en Python, on travaille avec des "objets".
- Les objets de type str ont de nombreuses fonctions prédéfinies qui leur sont spécifiques.
- Ces fonctions sont nommées des méthodes. Exemple avec la méthode replace :

```
x = "Salut à moi !"
x = x.replace("m", "t")
print(x)  #"Salut à toi !"
```

- Dans des cas de programmation réels, il ne faut pas hésiter à utiliser en priorité les méthodes déjà définies plutôt que de réinventer la roue.
- Dans ces TPs, on fera l'effort intellectuel d'imaginer que ces méthodes n'existent pas afin de comprendre leur fonctionnement interne !