Objets, variables et affectation

Ce cours est accompagné d'un notebook Capytale disponible aux élèves disposant d'un compte Toutatice.

1. Types de base

1.1. Objets et types de base

Le principe d'un programme est de manipuler des **données** avec des **instructions** afin de produire de nouvelles données.

En Python, des données sont appelées des **objets**, et tout ce qui est manipulable est un objet.

Tester les objets

Test1

Exécuter la commande suivante dans le terminal ci-dessous :

```
5 / 3
```

Test2

Exécuter la commande suivante dans le terminal ci-dessous :

```
"abra"+"cadabra"
```

Test3

Exécuter la commande suivante dans le terminal ci-dessous :

```
5<9
```

Test4

Exécuter la commande suivante dans le terminal ci-dessous :

```
4.5 + "3.2"
```

Une astuce

Dans un terminal, la seule ligne active est celle du **prompt**, c'est-à-dire la dernière marquée par les chevrons `>>>`. *On ne peut pas ré-exécuter une ligne déjà tapée*. Cependant, il est possible de récupérer une telle ligne, grâce à l'**historique de commande** obtenu en utilisant les flèches de direction `↑ Up` et `↓ Down`

```
>>>
```

Par exemple, dans le code suivant :

```
1 7+5
2 4.2+4.6
3 453*12
4 12-5
5
```

Lorsqu'on **exécute** ce code, on crée 4 objets différents, présents dans ce qu'on appelle l'**espace des objets**.

Cependant l'exécution du *script* ne renvoie aucune donnée (aucun *objet*). En effet, un mécanisme existant dans presque tous les langages de programmation, appelé **garbage collector** (soit *collecteur d'ordure*), nettoie automatiquement tout objet non utilisé. La mémoire de l'ordinateur étant limitée physiquement, il est nécessaire de nettoyer très régulièrement (plusieurs centaines de fois par seconde), afin de garantir le bon fonctionnement de la machine. Le garbage collector a donc supprimé les objets `12`, `8.8`, `5436` et `7`.

Regardons plus précisément ces objets, en demandant à Python grâce à l'instruction **built-in** `type()` la nature des objets créés.

Les types d'objets

Dans la console ci-dessous, testez une par une les commandes suivantes :

Test 1

```
type("Abracadabra")
```

Test 2

```
type(True)
```

Test 3

```
type(5)
```

Test 4

```
    type(4.5)
```

```
>>>
```

Une erreur ?

Types de bases et objets spécifiques

Les types basiques de Python sont donc les suivants :

- le type `int` : qui permet de manipuler des nombres entiers relatifs, de -2.10^9 à $+2.10^9$ (mais la plage pouvant être étendue, en pratique on ne s'intéresse pairs en python aux limites des entiers) ;
- le type `float` : qui permet de manipuler des nombres décimaux à 32 bits (s'écrivant avec 32 chiffres binaires) entre -10^{302} et $+10^{302}$ (*avec quelques subtilités, que nous verrons dans le chapitre correspondant au codage des nombres flottants*);
- le type `str` : qui permet de manipuler des chaînes de caractères ;
- le type `bool` : qui permet de manipuler des valeurs **booléennes**, c'est à dire vraies ou fausses, utilisées par défaut dans les instructions nécessitant une condition (`if` ou `while` par exemple).

Chacun de ces types possède des opérations qui lui sont propres. Elles sont nombreuses, et largement documentées dans la doc Python 3 officielle, en suivant le lien ci contre : Doc Python 3 en français.

Il existe aussi des *constantes* spécifiques, c'est-à-dire des objets spéciaux, comme par exemple `None`, qui est un objet sans type, signifiant une absence de valeur.

1.2. Nommage des objets et affectations

Bien entendu, de manière quasi obligatoire, il est nécessaire de conserver des objets et d'éviter que ceux-ci disparaissent avec le **garbage collector**.

Pour cela, on va associer à chaque objet un **nom** dans l'**espace de nom**, grâce à l'opération d'**affectation** `=`, comme par exemple `un_inconnu = "Toto"`. Ici le nom de variable `un_inconnu` est associé à l'objet de type chaîne de caractère `"Toto"`.

Une fois un objet associé à un nom, il n'est plus ramassé automatiquement par le *garbage collector*, et peut être rappelé plus tard dans le code en utilisant son nom.

Exemple

Exécutez le code suivant, puis testez dans la console à droite le type des objets suivants, en utilisant leur nom.

```
mon_entier = 5
le_vrai_hero = "Dark"
est_actif = True
mon_flottant = 3.2
c_est_lui = "Vador"

```

Testez ensuite chacune des propositions ci-dessous. Pour chacune d'entre elles, expliquez ce qui se passe.

1

```
mon_entier**2
```

2

```
le_vrai_hero+c_est_lui
```

3

```
mon_flottant if est_actif else mon_entier
```

4

```
mon_flottant if not(est_actif) else mon_entier
```

5

```
type(mon_flottant*mon_entier)
```

6

```
mon_entier == mon_flottant
```

7

```
le_vrai_hero*10
```

Comprendre l'affectation

Énoncé

On considère le code suivant :

```
a = 5
b = 4.2
c = a+b
a*c
a = a*3
b = 42
```

Compléter le tableau d'état des variables suivants en exécutant le programme à la main, puis vérifier dans la console :

a	b	c

Solution

ACADÉMIE DE RENNES Liberté Égalité Fraternité (http://www.ac-rennes.fr)

Français (FR)

Connexion (/accounts/login/?next=/video/23

ToutaPod (/)

Accueil (/) / Vidéos (/videos/) / Affectations de variables en Python et état mémoire

Règles de nommage des variables

Le nom donné à l'objet peut être n'importe lequel, en respectant les règles **impératives** suivantes :

- Un nom de variable est une séquence de lettres, de chiffres, **qui ne doit pas commencer par un chiffre**.
- La **casse est significative** (les caractères majuscules et minuscules sont distingués). Donc `Joseph`, `joseph`, et `JOSEPH` sont des variables différentes.
- Les **« mots réservés »** du langage sont déjà pris (ex : `type()`, `float`, `str`...). Il s'agit des instructions de bases et des fonctions natives (voir ici). Si vous nommez une variable comme un de ces mots, vous ne pourrez plus utiliser la fonctionnalité du mot réservé.

Pour les cracks

Conventions de nommage et PEP8

Il existe quelques règles de "bien nommage" des variables, définies dans ce qu'on appelle la PEP8, c'est-à-dire une description des bonnes pratiques d'écriture en Python. La PEP8 donne les conventions principales, qui permettent de lire plus facilement le code fourni par un·e autre codeur·euse. Les principales recommandations sont :

- Ne jamais utiliser les caractères "l" (L minuscule), "O" (o majuscule), ou "I" (i majuscule) seuls comme nom de variables. Dans certaines polices, ces caractères sont impossibles à distinguer des numériques un et zéro. Lorsque vous êtes tenté·e d'utiliser "l", utilisez "L" à la place.
- Il est recommandé d'écrire les noms de variables en minuscule. Si le nom de variable doit contenir plusieurs mots, on conseille d'écrire en **snake_case**, c'est-à-dire en séparant les mots par le caractère `_`. Exemple : `marge_brut`.
- Il convient aussi d'éviter autant que possible l'énumération de variables (`toto1`, `toto2`, `toto3`, ...), cela rend le programme parfaitement incompréhensible et sujet à des erreurs.
- De manière générale, il convient de suivre aussi strictement que possible les règles édictées par la **Python Software Foundation** à travers les normes PEP.

1.3. Exercices d'affectation

1. Écrire un code qui affecte l'objet `15` de type `int` à une variable nommé `valeur`, puis exécuter l'instruction `valeur*4`.

```
1
2
```

Evaluations restantes : 5/5

2. Écrire un code qui affecte l'objet `"15"` de type `str` à une variable nommé `texte`, puis exécuter l'instruction `texte*4`.

Evaluations restantes : 5/5

3. Compléter le code suivant afin que les objets des variables `a` et `b` soient inversés, **SANS UTILISER D'AFFECTATIONS DE NOMBRES, et sans toucher la valeur de `c`.**

```
1 a = 10
2 b = 200
3 c = 4000
4
```

Evaluations restantes : 5/5

4. Écrivez un code qui **permute** les objets des variables `a`, `b` et `c` (l'objet de `a` est affecté à `b`, l'objet de `b` est affecté à `c` et l'objet de `c` est affecté à `a`) **sans utiliser de valeurs numériques**.

```
a = 10
b = 200
c = 4000
```

Evaluations restantes : 5/5

2. Opérations sur les objets

2.1. Types numériques et opérations

Pour les types numériques, `int` et `float` (et pour le type `complex`, mais qui n'est pas vu en maths avant la terminale), on trouve toutes les opérations classiques.

Addition entière

```
>>> 5 + 2 #Addition entière
7
```

Addition flottante

```
>>> 5.6 + 3.4 # Addition flottante ( Quel est le type du résultat ?)
9.0
```

Addition mixte

```
>>> 5 + 3.4 # Addition entre un entier et un flottant ( Quel est le type du résultat ?)
8.4
```

Attention avec les flottants

Testez dans la console ci-dessous l'instruction suivante `0.1 + 0.2`. Qu'obtient-on ?

```
>>>
```

Soustractions

```
>>> 3.2-4 #Soustraction ( A tester de la même manière)
-0.79999999999
```

Multiplication

```
>>> 4*8.5 #Multiplication ( idem)
34.0
```

Division flottante

```
>>> 1/3 #Division
0.3333333333333333
```

Attention ! Au résultat ci-dessous, le type obtenu est `float`, même si le dividende et le diviseur sont entiers et que le résultat « tombe juste »...

```
>>> 4/2
2.0
```

Division Euclidienne

```
>>> 72//5 #Quotient de la division euclidienne
14
```

Ici par contre le résultat est bien de type `int` (A tester aussi...)

Reste de la Division Euclidienne

```
>>> 72%5 #reste de la division euclidienne ( ou modulo).
2
```

C'est un point important en informatique, nous avons souvent besoin du reste, aussi appelé **modulo**. Par exemple pour savoir si un nombre entier est pair, on utilise :

```
>>> 23%2 == 0
False
```

Puissances

```
>>> 10**7 #Puissance
10000000
```

Opérations avec les noms des objets

Tourtes les opérations faites avec les objets ci-dessus peuvent être effectuées directement sur les noms si ils existent :

```
>>> a = 5
>>> b = 2
>>> a + b
7
>>> a%b
1
```

Vous pouvez tester ces éléments dans le terminal ci-dessous :

```
>>>
```

2.2. Types `str` (chaines de caractères) et opérations

Déclarations

Une chaîne de caractère doit être déclarée :

- soit entre une paire de guillemets simples (*simple quote*) : `'Toto'` ;
- soit entre une paire de guillemets doubles (*double quote*) : `"Toto"`.

L'utilisation de l'un ou de l'autre n'a pas d'importance, mais on peut rapidement se tromper selon le contenu de la chaîne :

Tester les chaînes de caractères

Tester les chaînes suivantes dans le terminal ci-dessous:

```
1
texte = "Hello World !"
2
texte = 'Hello World !'
3
texte = "Salut l'ami !"
4
texte = 'Salut l'ami !'
5
texte = 'Alors là je dis : "Non !"'
6
texte = "Alors là je dis : "Non !""
```

```
>>>
```

Normalement, une chaîne de caractère, quelle que soit sa longueur, n'est considérée être que sur une seule et unique ligne. Il est cependant possible d'avoir des chaînes de caractères multi-lignes, à condition de les déclarer entre **trois paires** de guillemets identiques :

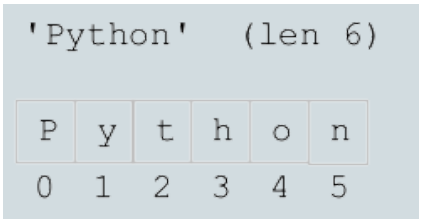
```
>>> zen = """
Préfère :
      la beauté à la laideur,
      l'explicite à l'implicite,
      le simple au complexe
      et le complexe au compliqué,
      le déroulé à l'imbriqué,
```

```
      l'aéré au compact.
Prends en compte la lisibilité.
Les cas particuliers ne le sont jamais assez pour violer les règles.
Mais, à la pureté, privilégie l'aspect pratique.
Ne passe pas les erreurs sous silence,
... ou bâillonne-les explicitement.
Face à l'ambiguïté, à deviner ne te laisse pas aller.
Sache qu'il ne devrait [y] avoir qu'une et une seule façon de procéder,
même si, de prime abord, elle n'est pas évidente, à moins d'être Néerlandais.
Mieux vaut maintenant que jamais.
Cependant jamais est souvent mieux qu'immédiatement.
Si l'implémentation s'explique difficilement, c'est une mauvaise idée.
Si l'implémentation s'explique aisément, c'est peut-être une bonne idée.
Les espaces de nommage ! Sacrée bonne idée ! Faisons plus de trucs comme ça.
"""
```

Indice d'un caractère

Chaque caractère d'une chaîne est repéré par son **indice**, c'est-à-dire par un **numéro donnant sa position**. Attention toutefois,en informatique, les indices commencent à 0 dans quasiment tous les langages !

Ainsi :



Dans la chaîne précédente, `"P"` à pour indice 0, `"y"` a pour indice 1 et `"n"` a pour indice 5.

Accès à un caractère par son indice

Les crochets

Vous utiliserez beaucoup les crochets en Python, pour les obtenir sur un clavier AZERTY :

- `[` s'obtient par `AltGr`+`[` ;
- `]` s'obtient par `AltGr`+`]` ;

On utilise l'indice entre crochets, juste après la chaîne de caractère.

```
>>> "Python"[3]
'h'
```

Si la chaîne est associée à un nom :

```
>>> texte = "DarkVador"
>>> texte[4]
'V'
```

Attention, si on cherche un indice qui n'existe pas, on a l'erreur suivante :

```
>>> "Abcdefg"[10]
Traceback (most recent call last):
    File "<input>", line 1, in <module>
IndexError: string index out of range
```

Cependant il est aussi possible de parcourir une chaîne à l'envers :

```
>>> "Python"[-1]
'n'
>>> "Python"[-3]
'h'
```

Longueur d'une chaîne

La longueur de la chaîne `"Python"`, c'est-à-dire **le nombre de caractères qui la composent**, est par contre bien de 6, et on peut y accéder grâce à la fonction **built-in** `len()`.

```
>>> len("Python")
6
```

La longueur d'une chaîne vide est 0 :

```
>>> len("")
0
```

Concaténation

La **concaténation** est l'opération consistant à accoler deux chaînes de caractères :

```
>>> "Toto"+"Tata"
'TotoTata'
```

On peut aussi concaténer une chaîne avec elle-même à plusieurs reprises :

```
>>> "Toto"*5
'TotoTotoTotoTotoToto'
```

2.3. Méthodes des chaînes de caractères

Le type `str` possède lui aussi ses propres fonctionnalités (ou **méthodes**) qui permettent de transformer ou modifier une chaîne de caractère. Parmi celles-ci, en voici quelques une utiles :

Substitution d'une sous-chaîne par une autre

```
>>> "abracadabra".replace("bra","BRUHHH")
"'aBRUHHHcadaBRUHHH'"
```

On utilise ici la **méthode** `replace` sur la chaîne de caractère `"abracadabra"`, et on va récupérer la nouvelle chaîne `'aBRUHHHcadaBRUHHH'`

Modifications en place ou non ?

Exécutez les lignes suivantes une par une dans le terminal ci-dessous :

```
>>> texte = 'abracadabra'
>>> texte.replace('a', 'U')
>>> texte
```

Le contenu de la variable `texte` n'a pas été remplacé. Un **nouvel objet de type `str`** a été créé, mais il a été immédiatement ramassé par le *garbage collector*, et a donc disparu. Il n'y a aucune modification de l'objet original qui est conservé (la modification **n'est pas en place**).Si on veux conserver :

- la chaîne originale et la chaîne modifiée, il faut donner un nouveau nom et affecter de la manière suivante :

###

- uniquement la chaîne modifiée, il suffit de réaffecter la modification au nom `texte` :

###

Changement de casse

Il existe les méthodes `.lower()` , `.upper()` et `.capitalize()` qui mettent respectivement la chaine originale en minuscule, en majuscule, et la première lettre en majuscule puis le reste en minuscule.

```
>>> "Toto".lower() #mise en minuscule, appel encore une fois à une méthode de classe.
'toto'
>>> "Toto".upper() #mise en majuscule, idem
'TOTO'
>>> "une phrase SANS QUEUE ni tÊte.".capitalize()
#Met en majuscule le premier caractère de la chaîne, et en minuscule les autres, idem
'Une phrase sans queue ni tête.'
```

Suppression des espaces inutiles avant et après une chaîne

Supprime les espaces inutiles devant et après une chaîne de caractères

```
>>> "              Des Blancs            ".strip()

'Des Blancs'
```

Séparation d'une chaîne selon un motif donné

Sépare une chaîne de caractère en, fonction d'une chaîne passée en argument, et renvoie une liste (que nous étudierons plus tard)

```
>>> "Une phrase est faite avec des mots".split("est")
['Une phrase ', ' faite avec des mots']
>>> "12:34:45:78".split(":")
['12', '34', '45', '78']
```

2.4. Les objets de type séquence

Il existe dans Python de nombreux autres types d'objets, et plus particulièrement les types `list` et `tuple`, que nous étudierons en détail plus tard dans l'année. Ces objets permettent de regrouper en une seule séquence ordonnée différents objets, et peuvent être alors utilisés simplement avec les possibilités suivantes (non exhaustives):

Construction d'une liste ou d'un tuple

Une liste se crée à l'aide des crochets `[` et `]`, avec deux possibilités :

- soit une liste vide :

  ```
  >>> ma_liste = []
  ```

- soit une liste contenant déjà des objets :

  ```
  >>> ma_liste = [5, 4.2, True, "Toto"]
  ```

Une liste en python est un objet qu'on peut modifier. On peut y ajouter un élément, en retirer un, en modifier un en particulier, etc.

Un tuple ne peut se crée à l'aide des parenthèses `(` et `)`, mais il doit déjà être rempli, car **il ne peut pas être modifié**(ni ajout, ni suppression, ni modification d'éléments) :

```
>>> mon_tuple = (12, 45, -13, "Bob")
```

Accès à un élément

Dans une liste ou un tuple, on peut accéder à un objet grâce à son **indice**, tout comme pour les chaines de caractères :

```
>>> ma_liste[3]
'Toto'
>>> mon_tuple[2]
-13
```

Ajouter un élément à une liste à la dernière position

```
>>> ma_liste = [5, 4.2, True, "Toto"]
>>> ma_liste.append(7)
>>> ma_liste
[5, 4.2, True, "Toto", 7]
```

Supprimer le dernier élément

```
>>> ma_liste = [5, 4.2, True, "Toto"]
>>> ma_liste.pop()
"Toto" # l'élément est renvoyé dans l'espace des objets courants
>>> ma_liste
[5, 4.2, True]
```

Modifier un élément d'indice donné :

```
>>> ma_liste = [5, 4.2, True, "Toto"]
>>> ma_liste[1] = "Abracadabra"
>>> ma_liste
[5, "Abracadabra", True, "Toto"]
```

3. Interactions avec l'utilisateur, transtypage des données, notions de formatage des chaînes de caractères

3.1. Afficher à l'écran

Jusqu'à présent, les seules choses que nous avons obtenu dans le terminal étaient des affichages simples des objets attachés à un nom connu. Il est possible de personnaliser, de différer, et de multiplier les affichages grâce à la fonction *built_in* `print()'.

Par exemple, essayez le code suivant :

```
###
```

La fonction `print` prend entre parenthèse un objet, et l'écrit sous la forme d'une chaine de caractères dans la console. Cependant on voit vite qu'en cas de multiples appels à la fonction `print`, on risque vite de se perdre dans les affichages. Il faut améliorer la sortie pour qu'elle soit compréhensible par un être humain.

Exécutez alors le code ci-dessous :

```
###
```

D'une part la fonction `print()` écrit les chaînes de caractères dans la console, mais en plus, grâce au `f` situé devant les chaînes comme :

```
f"La variable a contient {a}."
```

on a une chaine de caractère crée en remplacement le **nom** de la variable situé entre accolades par sa **valeur**. On appelle cela le **formatage des chaînes de caractères**, la documentation python vous donnera toutes les subtilités nécessaires. La sortie écrite par le programme dans la console est devenu bien plus compréhensible.

Les accolades

Vous utiliserez beaucoup les accolades en Python, pour les obtenir sur un clavier AZERTY :

- `{` s'obtient par `AltGr`+`'`;
- `}` s'obtient par `AltGr`+`=`;

3.2. Demander à l'utilisateur de saisir quelque chose au clavier

Pour demander une saisie clavier à un utilisateur, on utilise la fonction **built-in** `input()`, prenant *éventuellement* en **argument** une chaîne de caractères. Celle-ci interrompt le programme et attend une saisie clavier de l'utilisateur, et retourne cette saisie sous la forme d'une chaîne de caractère dès que la touche `Enter ⏎` est pressée.

```
###
```

Il faut cependant être attentif à ce qui est réalisé par la fonction `input()`. En effet, le **retour** effectué par cette fonction est renvoyé sous la forme de chaîne de caractères, ce qui peut poser un problème, comme par exemple dans la situation ci-dessous :

```
###
```

Pour lever cette ambigüité, nous sommes parfois obligés d'effectuer un **transtypage des données**, c'est-à-dire une modification du **type**, avec l'aide des fonctions **built-in** suivantes :

- `str()`
- `int()`
- `float()`
- `bool()`
- ...

Par exemple :

```
###
```

On a forcé ici dans la première ligne Python à transformer (si il le peut) le contenu de la variable `nb` comme étant un nombre entier.

Ceci ne lève cependant pas tous les problèmes, puisque si l'utilisateur·trice saisit une chaîne de caractères ne pouvant être transtypée en nombre entier, le programme renverra une erreur.

Exercice

Écrire dans l'éditeur ci dessous un code python qui :

- demande deux nombres entiers `a` et `b` à l'utilisateur·trice ;
- donne le produit des deux nombres, ainsi que le type du résultat, sous la forme d'une chaîne de caractères du type `3x4 = 12` (si l'utilisateur·trice à saisi 4 pour `a` et 3 pour `b`).

```
###
```