Numérique et science informatique

Lycée Hoche

année scolaire 2024-2025

Contents

1 Préambule

Dans toute la suite du cours ,le triple chevron >>> est **l'invite de commande** (prompt en anglais) de l'interpréteur Python:

```
>>> x=5
5
```

Sur une ligne , une suite de caractères précédés du signe # signifie que c'est un commentaire et ne fait donc pas parti du code python

```
>>> x=5 #Ceci est un commentaire
5
```

2 Objets , valeur et type

En `Python`, les données sont représentées sous forme d'objets. Toutes les données d'un programme `Python` sont représentées par des objets ou par des relations entre les objets (dans un certain sens, et en conformité avec le modèle de Von Neumann d'« ordinateur à programme enregistré », le code est aussi représenté par des objets). Chaque objet possède un *identifiant, un type et une valeur*.

```
>>> a=5
>>> print(id(a))
>>> 94612319838544
```

Le **type** de l'objet détermine les opérations que l'on peut appliquer à l'objet et définit aussi les valeurs possibles pour les objets de ce type. La fonction ***type()*** renvoie le type de l'objet (qui est lui-même un objet).

```
>>> mot='bonjour'
>>> type(mot)
<type 'str'>
```

Exercice

Tester dans l'interpréteur de python(**shell**) les exemples suivants en affichant le type de chacun des objets suivants:

2 ; 4.5 ; True ; (5,3) ; [4,-15,8] ,"hello world" .

3 les types de base

3.1 Les entiers

Il existe deux types d'entiers :

- Le type entier**(int)**

  Ils représentent les nombres, sans limite de taille, sous réserve de pouvoir être stockés en mémoire (virtuelle).Pour ce faire , on s'appuie sur la *représentation binaire* d'un entier naturel.L'étude de la représentation binaire sera vue ultérieurement.

```
>>> a=0
>>> type(a)
<type 'int'>
```

- Le type booléen(**boolean**)

  Ils représentent les valeurs **False** et **True**. Deux objets, False et True, sont les seuls objets booléens. Le type booléen est un sous-type du type entier et les valeurs booléennes se comportent comme les valeurs 0 (pour False) et 1 (pour True) dans presque tous les contextes. L'exception concerne la conversion en chaîne de caractères où "False" et "True" sont renvoyées.

3.2 Opérations sur les entiers

Les opérations binaires sur deux entiers :

opération	**symbole**	**type du résultat**
addition	$a + b$	int
multiplication	$a * b$	int
reste	$a\%b$	int
quotient entier	$a // b$	int
quotient	a/b	float
puissance	$a**b$	int (lorsque a et b sont positifs.)

3.3 Opérations sur les booléens

a	b	and	or
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Signalons également l'opération unaire *not* portant sur un booléen:

```
not(True)=False et not(False)=True
```

3.4 les décimaux (float)

Ce sont les nombres contenant un point décimal ou un exposant .
Ils représentent les nombres à virgule flottante en double précision, tels que manipulés directement par la machine. ceci dépend de l'architecture machine sous-jacente (et de l'implémentation C ou Java) pour les intervalles gérés et le traitement des débordements.

```
>>> x=5.3
>>> type(x)
<type 'float'>
```

ou encore:

```
>>> from math import *
>>> type(pi)
<type 'float'>
>>> 5e-4
0.0005
>>> type(8e-10)
float
```

Remarque importante: *Le type d'une variable est implicite: c'est l'affectation qui crée le type.*

Opérations sur les float

Les mêmes opérations listées ci-dessus pour les entiers (voir tableau) .
Dans la pratique , seules les opérations ayant un sens bien défini en mathématiques seront utilisées.Ainsi l'opération % n'a pas vraiment d'utilité lorsqu'elle porte sur deux décimaux.

Par ailleurs les règles de priorité sur les opérations binaires sont celles appliquées en mathématiques.

3.5 conversions de type

Il est possible de changer le type d'une variable :

```
>>> int(3.14)
3
>>> float(3)
3.0
```

Remarques:

- Python gère pleinement l'arithmétique de types numériques mixtes : lorsqu'un opérateur arithmétique binaire possède des opérandes de types numériques différents, l'opérande de type le plus « étroit » est élargi à celui de l'autre. Dans ce système, l'entier est plus « étroit » que la virgule flottante.

  ```
  >>> 2 + 3.14
  5.14
  ```

- Un autre type de nombres qu'on peut utiliser en Python : ce sont les nombres complexes.Il n'en sera pas question ici.

4 Autres types de variables:un premier exemple

4.1 Les chaînes de caractères

Il existe d'autres types de variables plus complexes (on parle de **types construits**)commes les variables de type **str** (chaînes de caractères), les variables de type **list** etc.
Une étude plus détaillée des listes (ainsi que des p-uplets et dictionnaires) sera faite au chapitre consacré aux types construits.
Nous aborderons ici la notion de chaînes de caractères et esquissons la notion de liste.

Chaîne de caractères (str)

Une chaîne de caractères en Python , type **str** , est une suite de caractères entourée par des guillemets " " ou par des guillemets simples ".

```
>>> 'arbre'
```

```
>>> "arbre"
```

Si le caractère " est présent dans la séquence on le précède d'un \

```
>>> 'grimper dans l\"arbre'
```

Pour concaténer des variables, comme par exemple des variables avec des chaînes littérales, utilisez l'opérateur + :

```
>>> 'deux' + 'deux'
'deuxdeux'
```

Exercice

1. Testez le code suivant:

   ```
   >>> "wagon"*"wagon"
   >>> "wagon"*3
   ```

2. Comment agit l'opérateur * sur les chaînes?

Les chaînes de caractères peuvent être indexées (ou indicées, c'est-à-dire que l'on peut accéder aux caractères par leur position), **le premier caractère d'une chaîne étant à la position 0**. Il n'existe pas de type distinct pour les caractères, un caractère est simplement une chaîne de longueur 1 :

```
>>> mot="bonjour"
>>> mot[0]
'b'
```

Les indices peuvent également être négatifs, on compte alors en partant de la droite. Par exemple :

```
>>> mot="bonjour"
>>> mot[-1]
'r'
```

En plus d'accéder à un élément par son indice, il est aussi possible de « trancher » (slice en anglais) une chaîne. Accéder à une chaîne par un indice permet d'obtenir un caractère, trancher permet d'obtenir une sous-chaîne :

```
>>> mot[1:4]   sous-chaîne à partir de la position 1 (incluse)  jusqu'à la position 4 (exclue)
>>> "onj"
```

Utiliser un indice trop grand produit une erreur :

```
>>> mot[10]  #la longueur de la chaine "mot" est 7
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Les chaînes de caractères, en Python, ne peuvent pas être modifiées. On dit qu'elles sont **immuables** ou **immutables**. Affecter une nouvelle valeur à un indice dans une chaîne produit une erreur :

```
>>> mot[1] = 'O'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

On peut parcourir les caractères d'une chaîne de deux façons:

1. En utilisant la fonction *range*:

   ```
   mot="alphabet"
   for i in range(len(mot)):
       print(mot[i], end = ' ')
   ```

A l'exécution, on obtient l'affichage sur la même ligne de tous les caractères en utilisant la fonction native **print** du langage Python.

```
>>> a l p h a b e t
```

Notez l'utilisation de la fonction native **len** qui s'applique sur une chaîne et qui renvoie la longueur de la chaîne (nombre de caractères)

2. En utilisant *in* :

```
mot="alphabet"
for c in mot:
    print(c, end = ' ')
```

Il existe des méthodes implémentées en Python et qui opèrent sur une chaîne **str**.
Nous ne ferons pas ici un catalogue complet de toutes les méthodes disponibles, mais on signale quelques unes comme **upper()** , **lower** , **strip()** ,**split** qui serons parfois utilisées dans les programmes que vous serez amenés à écrire.

Pour le lecteur désireux d'avoir un aperçu "complet" sur les méthodes de chaî,e en Python , nous renvoyons vers le lien:

https://docs.python.org/fr/3.9/library/stdtypes.html#string-methods

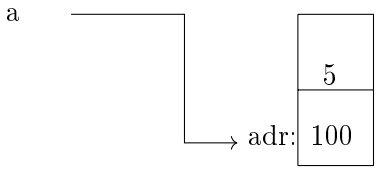
5 variables et affectation

5.1 Variable

Définition

Une variable est une zone de la mémoire de l'ordinateur dans laquelle une valeur est stockée.

L'affectation $a = 5$ signifie qu'à une certaine adresse (pour simplifier nous avons donné une étiquette à chaque adresse de la mémoire),on a stocké la valeur 5.

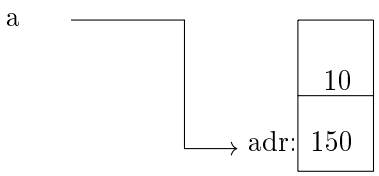


L'instruction x=a crée une variable x qui "pointe" vers la même adresse mémoire 100.

Testez le code suivant:

```
>>> a=5
>>> x=a
>>> a=10
>>> print(x)
```

L'affectation a=10 désigne une nouvelle adresse mémoire dont le contenu est la valeur 10 .



5.2 Noms des variables

• Pour nommer une variable on peut utiliser des lettres minuscules ou majuscules ou le caractère souligné (_):

```
>>> voiture_sport='mazerati'
```

• On peut également utiliser les chiffres , mais **il est interdit** d'utiliser un chiffre au début du nom d'une variable.
• Vous ne pouvez pas utiliser d'espace dans un nom de variable.
• On évite également les mots réservés de `Python` (comme *print* , *int* , *for* etc) comme nom de variable.
• Le nom d'une variable est sensibles à la casse :

```
>>> aire=5.5
>>> Aire=6.8
```

aire et Aire sont des identificateurs distincts pour Python. En informatique, les noms donnés aux variables sont souvent désignés sous le terme *identificateur*. Dans les exemples précédents, **a** et **x** sont des identificateurs de variables.
Chaque langage de programmation établit des règles déterminant quels sont les identificateurs acceptés. `Python` n'échappe pas à la règle.

6 instructions et expressions

L'affectation x = x + 3 est une instruction. Elle est composée d'une variable x et d'une *expression* x + 3 .
On attribue une valeur à chaque expression.
Pour les expressions sans variables, comme 7 *2 + 7 dont la valeur est 21, la valeur s'obtient simplement en effectuant les opérations présentes dans l'expression, dans cet exemple une addition et une multiplication.
La valeur d'une expression qui contient des variables, par exemple (2 + x) * 3 , se définit de la même manière, mais dépend de l'état dans lequel on calcule cette valeur.
L'évaluation d'une expression produit un objet du type donné (chaîne de caractères, chaîne d'octets, entier, nombre à virgule flottante, nombre complexe) avec la valeur donnée.

7 Les blocs d'instruction

NB: Dans toute cette partie nous nous sommes inspirés pour les définitions et les exemples du site[1].

7.1 Les blocs if

La forme générale du bloc if :

```
if expression1 :
   #un bloc d'instruction 1

elif expression2 :
   #un autre bloc d'instruction 2
elif expression3 :
   #un autre bloc 3
   #On peut mettre plusieurs "elif:"
   #Avec d'autres blocs ..

else:
   #un dernier bloc d'instructions
```

D'ores et déjà, notez la présence des deux-points « : » à la fin de la ligne 1 débutant par if.
Cela signifie que l'instruction if attend un bloc d'instructions.
Comment indique-t-on à Python où ce bloc commence et se termine ?

Cela est signalé uniquement par l'indentation, c'est-à-dire le décalage vers la droite de la (ou des) ligne(s) du bloc d'instructions.

La suite d'instuctions sélectionne exactement un des blocs en évaluant les expressions une par une jusqu'à ce qu'une soit vraie (voir la section Opérations booléennes pour la définition de vrai et faux) ; ensuite cette suite est exécutée (et aucune autre partie de l'instruction if n'est exécutée ou évaluée).

Si toutes les expressions sont fausses, la suite de la clause else, si elle existe, est exécutée.

7.2 La boucle for

l'instruction **for** en Python itère sur les éléments d'une séquence (qui peut être une liste, une chaîne de caractères...), dans l'ordre dans lequel ils apparaissent dans la séquence.(Voir[1])

```
Villes=['Lille' , 'Bordeaux', 'Nantes', 'Marseille']
for v in Villes:
    print(v,len(v))
```

On obtient à l'exécution:

```
Lille 5
Bordeaux 8
Nantes 6
Marseille 9
```

La variable v est appelée variable d'itération, elle prend successivement les différentes valeurs de la liste Ville à chaque itération de la boucle.
On peut choisir le nom que l'on veut pour cette variable (à l'exception des mots réservés).

Celle-ci est créée par Python la première fois que la ligne contenant le for est exécutée (si elle existait déjà son contenu serait écrasé).

Une fois la boucle terminée, la variable d'itération v ne sera pas détruite et contiendra la dernière valeur de la liste Ville.

7.3 Utilisation de la fonction range()

Si vous devez itérer sur une suite de nombres, la fonction native *range*() est faite pour cela. Elle génère des suites arithmétiques :

```
>>>for i in range(4):
......print(i**2)
0
1
4
9
```

forme générale de la boucle for utilisant un indice:

```
for i in range(a1,a2, pas):
    ........
```

- le premier indice de la boucle est égal à $a1$.
- le dernier indice vaut $a2 - 1$
- pas est le "pas" (qui peut être négatif le cas échéant)

Exemples

Testez les exemples suivants dans le shell.

1. ```
   for i in range (7,2,-1):
       print("foo")
   ```
2. ```
   for i in range (7,2):
       print("foo")
   ```

7.4 La boucle while

L'instruction **while** est utilisée pour exécuter des instructions de manière répétée tant qu'une *condition* est vraie :

```
while (condition):
    #bloc d'instructions
    #...
    #...
```

Cette condition prendra souvent la forme d'une *expression* portant sur une variable qui sera incrémentée dans le bloc d'instructions (c'est-à-dire qu'on lui rajoutera 1 à chaque exécution du bloc).

Tant que la valeur de *expression* est vrai (valeur True pour Python), le bloc d'instruction s'exécute à nouveau.

```
n=0
while n<4:
    #Un bloc d'instructions
    print(n**2)
    n=n+1
```

Ici la condition porte sur la valeur de la variable n qui doit donc être initialisée avant l'entrée dans la boucle.
De plus sans l'incrémentation , $n = n + 1$, la valeur de la variable reste toujours à sa valeur initiale.
La condition pour rentrer dans la boucle est donc toujours remplie.
En conséquence, la boucle tournera indéfiniment. On a alors créé une boucle infinie.
A l'exécution:

```
0
1
4
9
```

8 Les fonctions

8.1 Ecrire des fonctions

Le langage `Python` possède de nombreuses fonctions qui ont été implémentées par les concepteurs du langage (Le créateur du langage Guido Van Rossum érivit une première version en 1989).

Considérons la suite d'instructions exécutées par un robot qui doit conduire une des dix voitures , portant les étiquettes de A à J , stationnées dans un parking:

- Ouvrir la portière de la voiture A.

- s'installer au volant.
- Mettre le contact.
- Enclenchez la première.

On peut supposer que le robot doit répéter ces instructions plusieurs fois au cours d'un trajet.
D'où l'idée de regrouper toutes ces instructions dans une *fonction demarrer()*.

En python la définition de cette fonction est donnée par :

```
def demarrer():
    #Ouvrir la portiere de la voiture A.
    #s'installer au volant.
    #Mettre le contact.
    #Enclenchez la premiere.
```

Supposons que dans un lot de 10 voitures , le robot doit démarrer la voiture E plutôt que la voiture B , il suffit alors d'indiquer à la fonction son numéro lors de son appel.
demarrer('E').

En python la définition de cette fonction est donnée par :

```
def demarrer(ETIQUETTE):
    #Ouvrir la portiere de la voiture portant le numero ETIQUETTE.
    #s'installer au volant.
    #Mettre le contact.
    #Enclenchez la premiere.
```

On dit que ETIQUETTE est un **paramètre** (ou argument formel) de la fonction demarrer.
Lors de l'*appel* de la fonction : demarrer('A') , A est l'argument *réel* de la fonction *demarrer*

Supposons à présent que le robot , lors d'une séance d'essais , doive prendre un certain nombre de mesures:l'état d'usure des freins par exemple.
Une fonction Etat_Freins() peut alors contenir un certain nombre d'instructions avec en plus le renvoi d'un résultat : celui de l'état des freins (par exemple un entier allant de 0 à 10).

```
def Etat_Freins(Etiquette):
    #instruction 1
    #instruction 2.
    #...
    return 8
```

A la ligne 5 , on a utilisé le mot clé **return** pour indiquer que la fonction Etat_Freins() retourne une valeur.
Notez que la fonction *demarrer()* ne retournait pas de valeur.On dit parle alors de **procédure**.

Valeurs par défaut des arguments

La forme la plus utile consiste à indiquer une valeur par défaut pour certains arguments. Ceci crée une fonction qui peut être appelée avec moins d'arguments que ceux présents dans sa définition.

Par exemple :

```
def Etat_Freins(ETIQUETTE,vitesse=80):
    #instructions
    #....
    if vitesse > 110:
        return 3
    else:
        return 8
```

)

L'appel Etat_Freins('A') entraîne comme résultat 8

```
>>>Etat_Freins('A')
8
```

L'appel Etat_Freins('A',120) entraîne comme résultat 8

```
>>>Etat_Freins('A')
>>> 3
```

Variables locales et variables globales au sein d'une fonction

Les variables utilisées dans une fonction sont de deux sortes : **locales** ou **globales**.

Les arguments formels d'une fonction sont locales ainsi que les variables employées dans la fonction qui ne sont pas explicitement déclarées globales. Après l'exécution d'une fonction , les variables locales sont "détruites".

Une variable est globale si dans une fonction elle a été déclarée comme telle ou si cette variable était déjà affectée avant l'appel de la fonction.

```
TVA=10
def Valeur_TTC(prix):

    prix_TTC=prix*(1+TVA/100)
    return prix_TTC
```

Dans l'espace de la fonction Valeur_TTC , la variable prix_TTC est locale. Après la fin de l'exécution de la fonction Valeur_TTC , l'instruction print(prix_TTC) provoque un message d'erreur:

```
>>> print(prix_TTC)
NameError: name 'prix_TTC' is not defined
```

Si la fonction Valeur_TTC() doit utiliser une autre valeur que 10 pour la TVA, alors l'on pourrait être tenté d'écrire:

```
TVA=10
def Valeur_TTC(prix):
    TVA=5
    prix_TTC=prix*(1+TVA/100)
    return prix_TTC
```

A l'exécution, c'est correct:

```
>>> print(Valeur\_TTC(100))
105
```

Mais si nous écrivons ensuite :

print(100*(1+TVA/100))
Nous obtenons 110. Que s'est-il passé? Dans le corps de la fonction Valeur_TTC, le prix TTC a été calculé en utilisant la valeur 5 pour TVA.
Après son exécution , TVA a toujours sa valeur initiale , celle d'avant l'appel de la fonction.

Une solution est de déclarer TVA comme **global** à l'intérieur de la fonction :

```
TVA=10
def Valeur_TTC(prix):
    global TVA
    TVA=5
    prix_TTC=prix*(1+TVA/100)
    return prix_TTC
```

Exercice

Testez les exemples suivants dans le shell.

```
def f():
    print("bonjour")
    return
```

La fonction précédente se contente d'afficher le mot "bonjour". Elle ne renvoie aucun résultat malgré la présence du mot réservé *return*. Il est recommandé pour prendre dès maintenant les bonnes habitudes de mettre un "*return*" à la fin du code de toute fonction.

```
def carre(x):
    return x**2
```

La fonction précédente calcule le carré d'un nombre x donné en **argument** de la fonction et retourne le résultat.

```
>>>def carre(x):
...t=x**2
>>>print(carre(2))
none
```

Dans l'exemple précédent la fonction calcule le carré de x mais ne renvoie pas de valeur.

Exemples d'utilisation de la fonction native **print**()

```
>>> x=5
>>> y=70
>>> print (x,y)
5 70
```

```
>>> print("une valeur approchée du nombre pi est " , 3.14)
une valeur approchée du nombre pi est  3.14
```

Dans les exemples précédents les valeurs sont affichées sur la même ligne séparées par un espace. Plus généralement si on souhaite afficher les deux valeurs sur la même ligne en utilisant un autre séparateur :

```
>>>x=5
>>>y=70
>>>print (x,y,sep= " ; ")
5 ; 70
```

8.2 La spécification

Toute fonction doit être documentée (en tout cas , il est fortement recommandé de le faire afin de faciliter sa maintenance.)

Voici un modèle d'écriture ci-dessous (ce n'est pas le seul).
Des informations sont inscrites entre "' ... "'.

On appelle ce procédé la **spécification** .

```
def nomFonction(parametre1, parametre2):

    """
    Entrées:
    parametre1 - type du parametre, precondition eventuelle
    parametre2 - type du parametre, precondition eventuelle

    Sortie: type de ce qui est renvoye - liens avec les parametres
     ou description de l'affichage realise
    """
    # corps de la fonction
    #bloc instructions
    ..................
    ..................

    return Sortie  #ou return si la fonction ne renvoie pas de valeur
```

Dans la spécification ,les "préconditions" sont des conditions à respecter par les paramètres d'une fonction pour que les instructions effectuées par cette fonction aient un sens. Ces préconditions sont données par le cahier des charges du programme et sont rappelées dans le *docstring*.
Ce *docstring* présente au moins la forme minimale donnée ci-dessus. Il spécifie la fonction, c'est-à-dire :

- décrit les paramètres (leur type) ;
- les préconditions éventuelles sur ces paramètres ;
- le type de la sortie ainsi que son lien avec les paramètres.

8.3 Mise au point à l'aide de tests simples: utilisation de assert

L'idée la plus simple à mettre en place est d'utiliser la commande **assert** pour vérifier que les résultats voulus correspondent à ce que renvoie la fonction désirée. En particulier :
Exemple :

```
from math import sqrt
def racine_carree(x):
    '''
    entrée:
    x: -int ou float , x positif ou nul

    Sortie : -float , racine carrée de x
    '''
    return sqrt(x)

assert racine_carree(0)== 0
assert racine_carree(4) == 2.0
assert racine_carree(-4) == 2 , 'la fonction ne prend pas comme paramètre
                        un nombre négatif'
```

Notez la possobilité d'utiliser un commentaire en cas d'échec (dernière ligne)

8.4 le module doctest

Le module doctest permet d'inclure les tests dans la docstring descriptive de la fonction écrite.
On présente dans la docstring,en plus des explications d'usage, des exemples d'utilisation de la fonction tels qu'ils pourraient être tapés directement dans la console Python. Le module doctest(via l'appel à doctest.testmod()) reconnaît les bouts de code correspondant à ces exemples et les exécute pour les comparer à la sortie demandée.

On peut alors commenter in situ les tests et leurs raison d'être et avoir une sortie détaillée et globale des tests qui ont réussi ou raté.

Se reporter aux exemples donnés dans les TP (exercices)

Bibliography

[1]

[2]