



Cours : Python

Algorithme vs Programme

Algorithmie

- Solution « informatique » relative à un problème
- Suite d'actions (instructions) appliquées sur des données
- 3 étapes principales :
 1. saisie (réception) des données
 2. Traitements
 3. restitution (application) des résultats

Programme

- Transcription d'un algorithme avec une syntaxe prédéfinie
- **Python**
- Même principes fondamentaux que les autres langages objets (Delphi, Java, C#, etc.)
- Python s'enrichit de bibliothèques de calcul spécialisées (mathématique, bio informatique, etc.)

Modulable

Python permet de séparer les programmes en modules qui peuvent être réutilisés dans d'autres programmes en Python

Orienté Objet

Syntaxe aisée

La syntaxe de Python est très simple et, combinée à de nombreux types de données évolués (comme les listes, dictionnaires, tuples...), ce qui conduit à des programmes à la fois très compacts et très lisibles. De plus, Python ne nécessite aucune déclaration de variable. Les variables sont créées lors de leur première assignation.

Un petit historique...

❑ En 1989, **Guido Van Rossum** commença à travailler sur Python qui n'était alors qu'un projet lui servant d'occupation durant les vacances de Noël pendant lesquelles son bureau était fermé.

- Le but de Guido était d'inventer un successeur **au langage ABC**, un langage d'apprentissage peu apprécié dans le milieu académique.
- il fait appel directement à des utilisateurs Unix habitués au langage C. il a voulu que Python soit facilement utilisable dans d'autres langages et environnement contrairement à ABC. Il y réussit globalement...

Aperçu de ses caractéristiques...

Langage Script

Tout comme Perl, Tcl et Rexx, Python fait partie des **langages script interprétés** contrairement à C/C++ qui sont des langages compilés □ plus rapide au développement, de comporter moins de ligne (50% de moins). Par contre, Il est plus lent à l'exécution

Portable

Python est portable entre les différentes variantes de Unix ainsi que sur les OS propriétaires comme Mac OS, MS-DOS et les différentes versions de Windows

Gratuit

Python est placé sous Général Public License. Il est facilement téléchargeable sur www.python.org

Extensible

Au-delà de la multitude de librairies et de modules déjà existante, il est possible d'en développer pour ses propres besoins

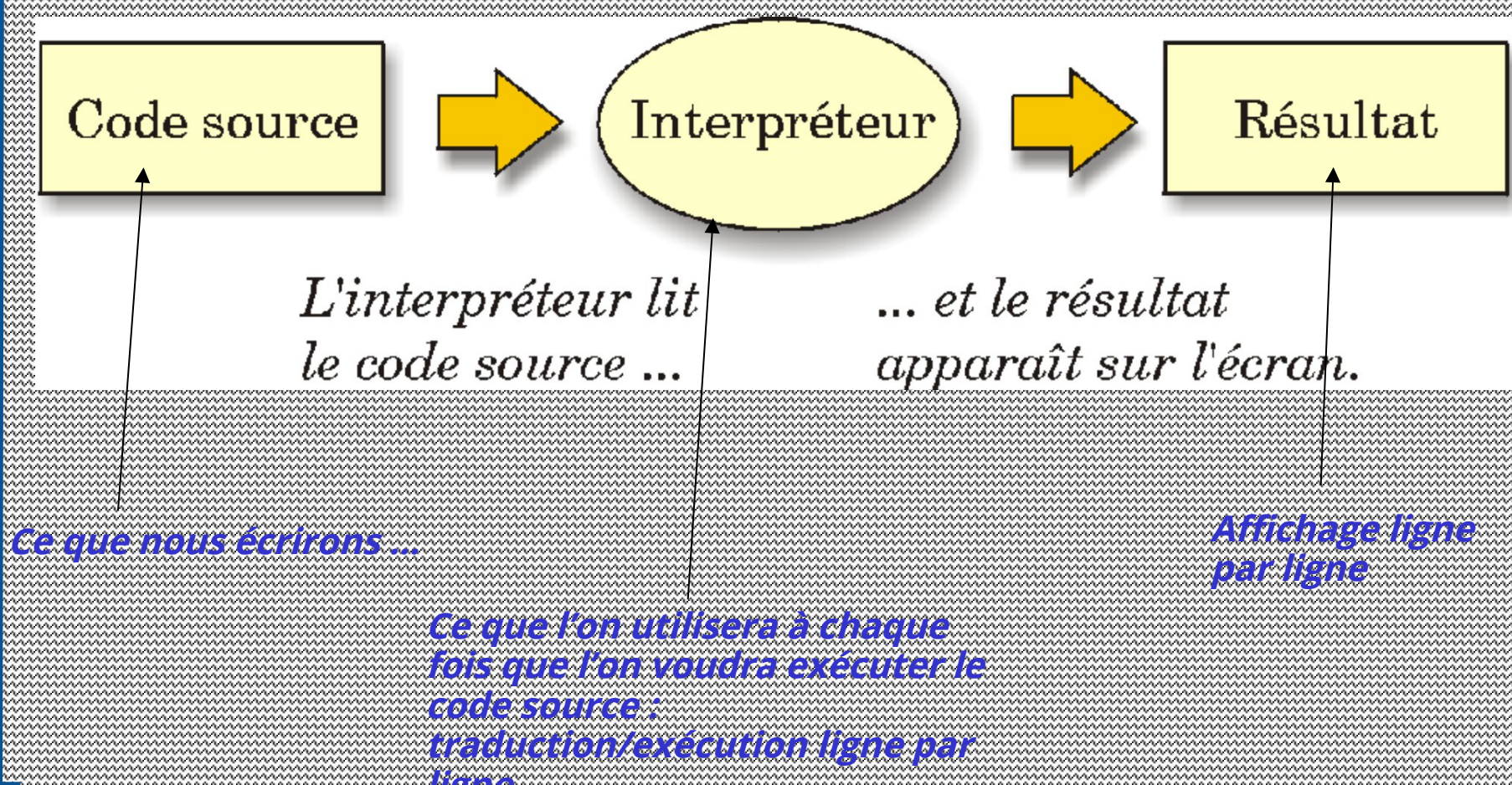
Domaines d'application

- **Scripts d'administration systèmes**
ex : Les programmes d'administration système spécifiques à la distribution Red Hat Linux.
- **Tous les développements lié à l'internet** et en particulier au web (moteur de recherche, navigateur...)
ex : moteurs de recherche yahoo et infoseek
- **Accès aux bases de données** (contextuelle)
- **Réalisations d'interfaces graphiques utilisateurs.**
- Utilisation pour ***la résolution de calculs scientifiques***
ex : Python est notamment utilisé pour les fusées de la NASA
- ...

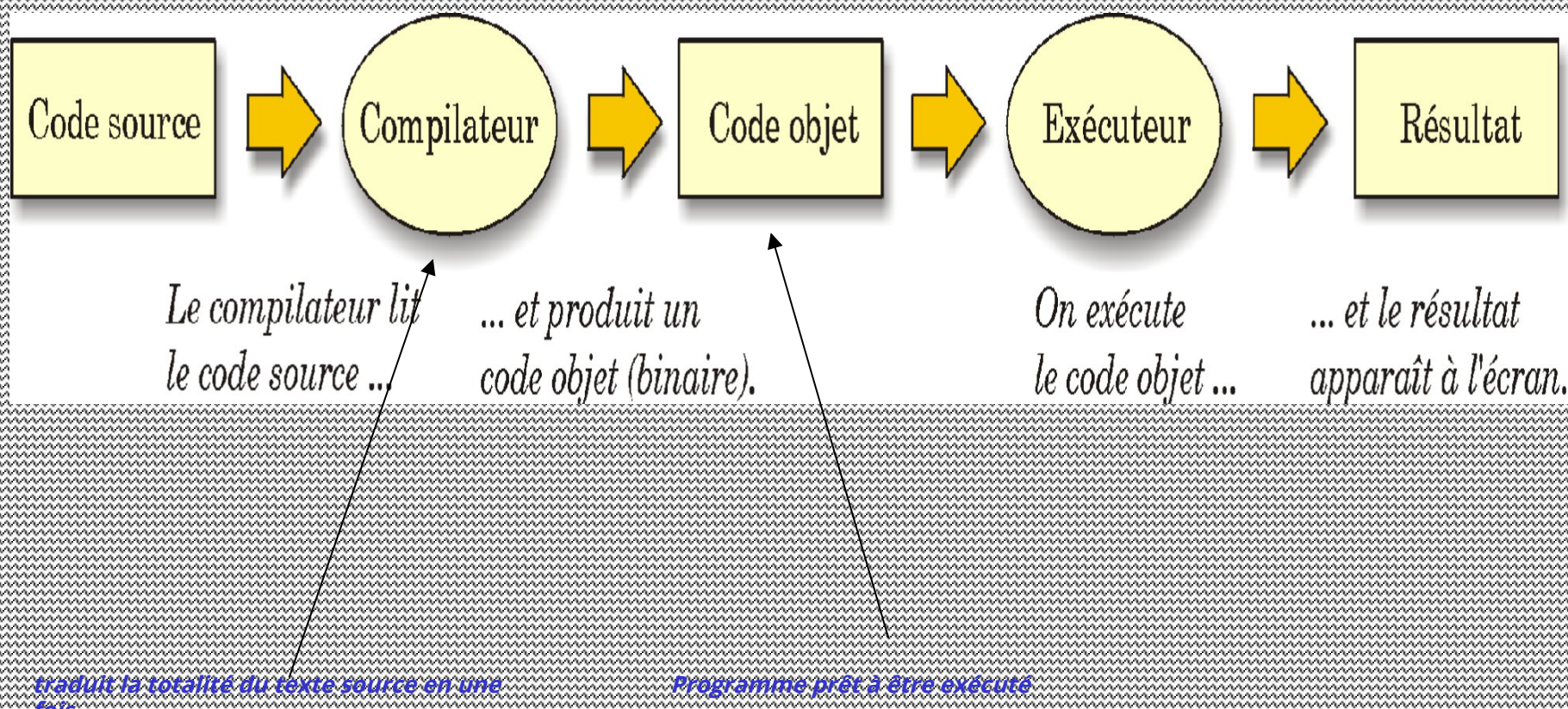
Langage machine, langage de programmation

- Il s'agit d'un langage de haut niveau
- Il est beaucoup plus facile d'écrire un programme dans un langage de haut niveau : l'écriture du programme prend donc beaucoup moins de temps (versus sa traduction)
- le programme sera souvent portable : on peut le faire fonctionner sans aucune modification, sur des machines ou des systèmes d'exploitation différents (versus programme de bas niveau).

Compilation et interprétation



Compilation et interprétation



traduit la totalité du texte source en une fois

Programme prêt à être exécuté

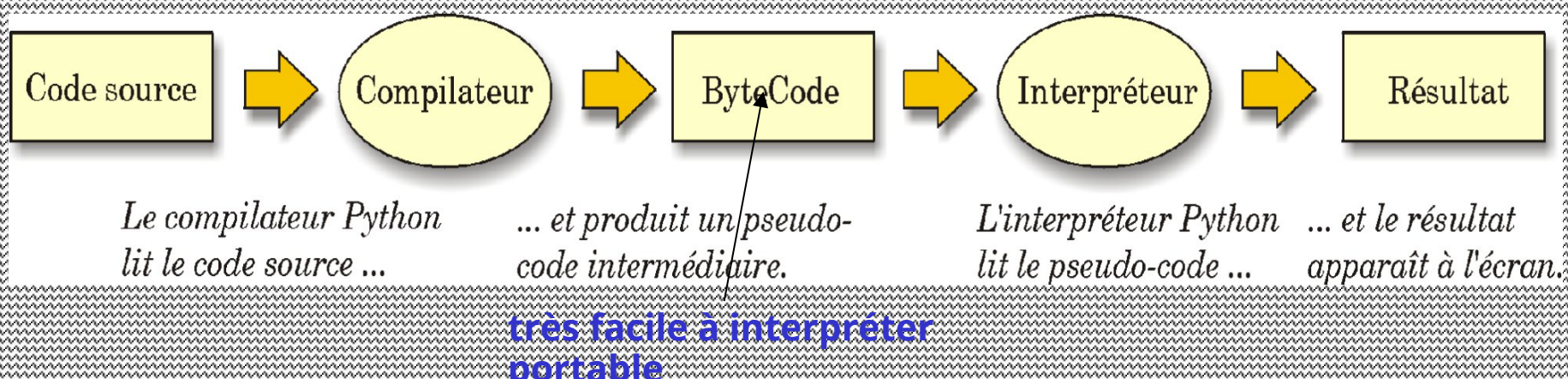
Compilation et interprétation

Certains langages modernes tentent de **combinaison** **les deux techniques** afin de garder le meilleur de chacune.

- Python



- JAVA



Remarque

- Tout ceci peut paraître un peu compliqué, mais la bonne nouvelle est que **tout ceci est pris en charge automatiquement par l'environnement** Python.
- Il vous suffira d'entrer vos commandes au clavier, de frapper <Enter>, et Python se chargera de les compiler et de les interpréter pour vous.

Mise au point d'un programme. Recherche des erreurs «debug»

3 types d'erreurs :

1. **syntaxe** : se réfère aux règles que les auteurs du langage ont établies pour la structure du programme.
2. **sémantique** : c'est une erreur de logique, i.e, le programme est sans erreurs mais les résultats sont inattendus.
3. **d'exécution ou « Run-time error »** : lorsque votre programme fonctionne déjà, mais que des circonstances particulières se présentent (par exemple, votre programme essaie de **lire un fichier qui n'existe plus, une division par zéro**).



Python propose les outils standards de programmation(1/1)



Données typées. Python propose les types usuels de la programmation : entier, réels, booléens, chaîne de caractères.

Structures avancées de données. Gestion des collections de valeurs (énumérations, listes) et des objets structurés (dictionnaires, classes)

Séquences d'instructions, c'est la base même de la programmation, pouvoir écrire et exécuter une série de commandes sans avoir à intervenir entre les instructions.

Structures algorithmiques : les branchements conditionnels et les boucles.



Python propose les outils standards de programmation(1/2)



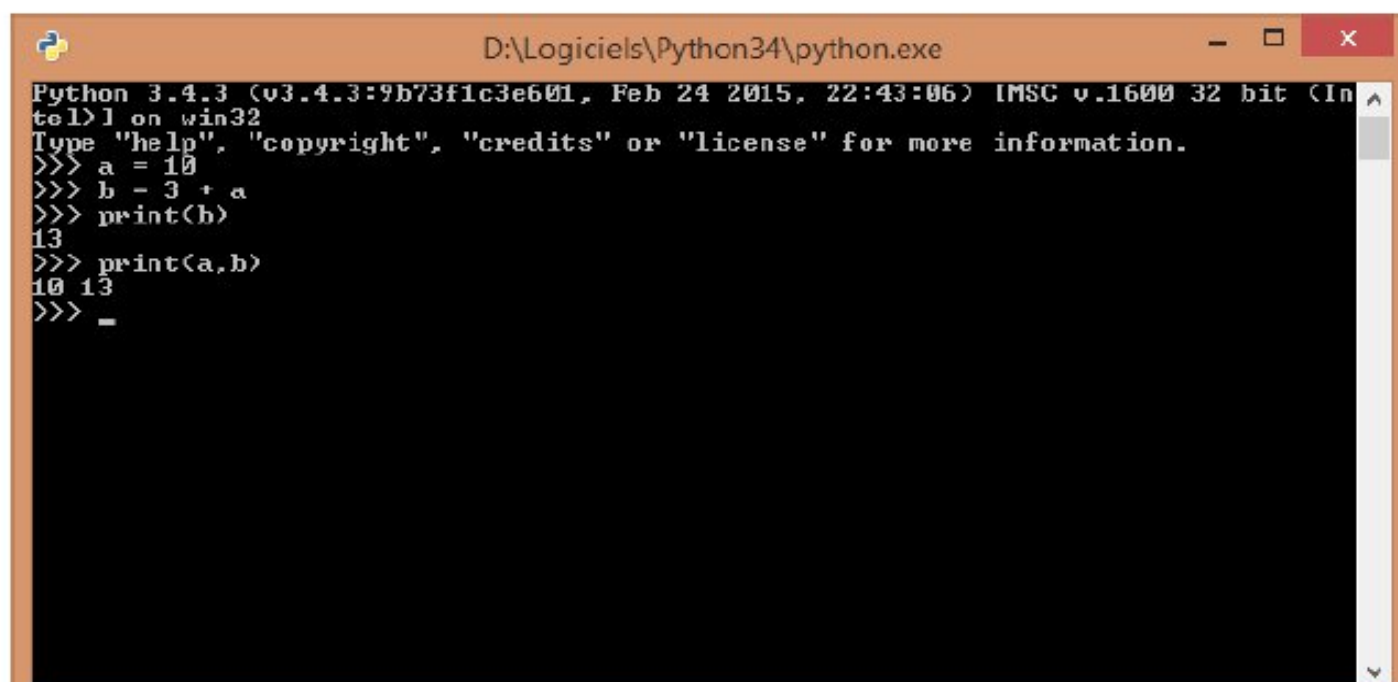
Les outils de la programmation structurée : pouvoir regrouper du code dans des **procédures** et des **fonctions**. Cela permet de **mieux organiser** les applications.

Organisation du code en **modules**. Fichiers « **.py** » que l'on peut appeler dans d'autres programmes avec la commande **import**

Possibilité de distribution des modules : soit directement les fichiers « **.py** », soit sous forme d'extensions prêtes à l'emploi.

Python est « **case sensitive** », il différencie les termes écrits en minuscule et majuscule. Des conventions de nommage existent Mais le plus important est d'être raccord avec l'environnement de travail dans lequel vous opérez.

Python mode opérateur 1



```
D:\Logiciels\Python34\python.exe
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> a = 10
>>> b = 3 + a
>>> print(b)
13
>>> print(a,b)
10 13
>>> _
```

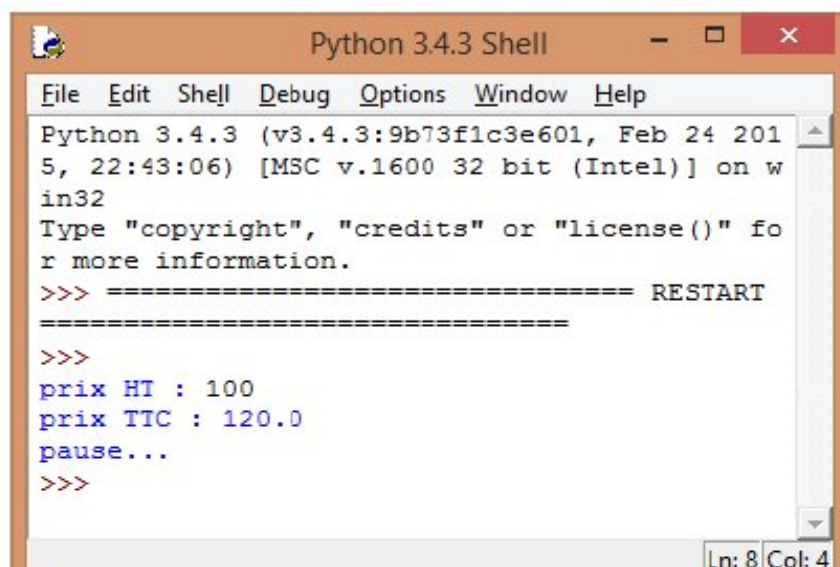
Lancer la console Python et introduire les commandes de manière interactive.

→ Ce n'est pas adapté pour nous (programmation = enchaînement automatique d'instructions)

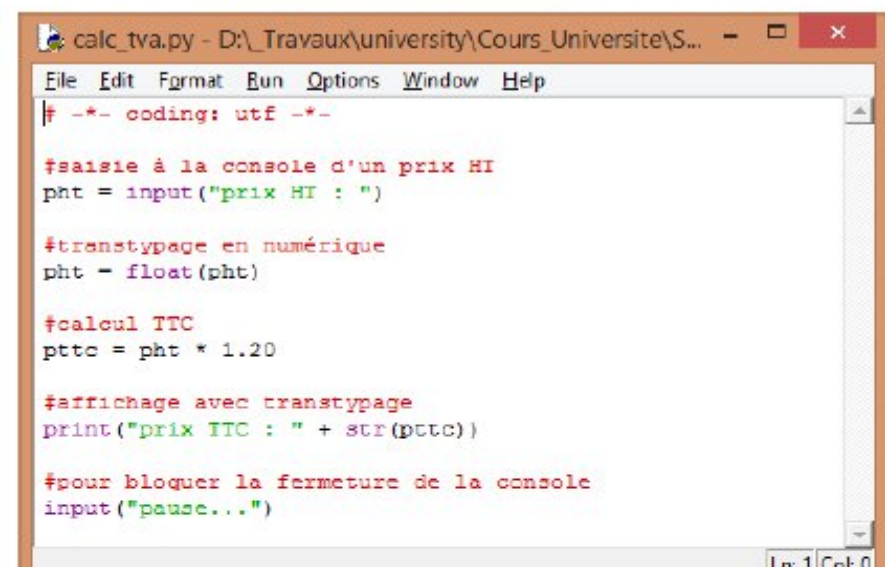
Python mode opérateur 2

Shell : fenêtre d'exécution du programme

Editeur de code



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART
>>>
prix HT : 100
prix TTC : 120.0
pause...
>>>
```



```
calc TVA.py - D:\Travaux\university\Cours_Universite\S...
File Edit Format Run Options Window Help
# -*- coding: utf-8 -*-
#saisie à la console d'un prix HT
pht = input("prix HT : ")

#transpage en numérique
pht = float(pht)

#calcul TTC
pttc = pht * 1.20

#affichage avec transpage
print("prix TTC : " + str(pttc))

#pour bloquer la fermeture de la console
input("pause...")
```

Menu : RUN / RUN MODULE
(ou raccourci clavier F5)

Permet de mieux suivre l'exécution du programme. Messages d'erreur accessibles, pas comme pour l'exécution console.

Première opération

Affectation – Typage automatique

➤ `a = 1.2`

a est une variable, en interne elle a été automatiquement typée en flottant « float » parce qu'il y a un point décimal. **a** est l'identifiant de la variable (attention à ne pas utiliser les mots réservés comme identifiant), **=** est l'opérateur d'affectation

Calcul

➤ `d = a + 3`

d sera un réel contenant la valeur 4.2

Forcer le typage d'une variable (sert aussi pour le transtypage)

➤ `b = float(1)`

Même sans point décimal, **b** sera considéré comme float (**b** = 1, il aurait été int dans ce cas).

Connaître le type d'un objet

➤ `type(nom_de_variable)`

Affiche le type interne d'une variable (ex. `type(a)` → `<class 'float'>`)

Supprimer un objet de la mémoire

➤ `del nom_de_variable`

où `nom_de_variable` représente le nom de l'objet à supprimer.

Affectation

Affectation simple

La seconde évite les ambiguïtés.

```
#typage automatique  
a = 1.0  
#typage explicite  
a = float(1)
```

Affectations multiples

Pas fondamental

```
#même valeur pour plusieurs variables  
a = b = 2.5  
  
#affectations parallèles  
a, b = 2.5, 3.2
```


Affectations multiples

Sous Python, on peut assigner une valeur à plusieurs variables simultanément.

Exemple :

```
>>> x = y = 7
```

```
>>> x
```

```
7
```

```
>>> y
```

```
7
```

On peut aussi effectuer des *affectations parallèles* à l'aide d'un seul opérateur :

```
>>> a, b = 4, 8.33
```

```
>>> a
```

```
4
```

```
>>> b
```

```
8.33
```

Dans cet exemple, les variables **a** et **b** prennent simultanément les nouvelles valeurs 4 et 8,33.

Opérations, expressions enchaînements

La plus couramment utilisée

1 instruction = 1 ligne

```
a = 1  
b = 5  
d = a + b
```

Autres possibilités

Personne n'utilise
ces écritures

```
a = 1;b = 5 ;d = a + b;
```

```
a = 1;  
b = 5;  
d = a + b;
```

Une opération particulière

Une variable ne se comporte pas
de la même manière de part et
d'autre du symbole d'affectation

```
a = 2  
a = a + 1
```


La présentation des programmes

Les commentaires

Un programme source est destiné à l'être humain.

- Pour en faciliter la lecture, il doit être judicieusement commenté.
- La signification de parties non triviales doit être expliquée par un commentaire.
- Un commentaire commence par le caractère # et s'étend jusqu'à la fin de la ligne :

```
#-----  
# Voici un commentaire  
#-----
```

```
n = 9 # En voici un autre
```

Les variables

Variable

= conteneur d'information qui porte un nom

= référence à une adresse mémoire (informatiquement)

Les noms des variables sont conventionnellement écrits en minuscule.

Ils commencent par une **lettre** ou le caractère **_**,
puis éventuellement, des lettres, des chiffres.

La casse est significative
(les caractères majuscules et minuscules sont distingués).

Ils doivent être **différents** des mots réservés de Python.

Les variables (suite)

Mots réservés : On ne peut pas utiliser comme nom les 33 mots suivants (qui sont réservés au langage Python)

and	as	assert	break	class	continue	def
del	elif	else	except	False	finally	for
from	global	if	import	in	is	lambda
None	nonlocal	not	or	pass	raise	return
True	try	while	with	yield		

Les variables (suite)

Conventions sur les noms en général

- But : donner de la lisibilité sur les noms et limiter les disfonctionnements entre les différentes plates-formes (Unix, Window, Macintosh).
 - Jamais de caractères accentués
 - Jamais de blanc entre les mots

Syntaxe proposée et appliquée

- Variable : `nomDeVariable`
- Sous-programme : `nomFonction(..)`

Exemple

- Variable : `valMin`
- Sous-programme : `ecrireChaine(...)`

Les variables (suite)

Typage des variables (spécifique à Python)

- il n'est pas nécessaire de définir le type des variables avant de pouvoir les utiliser.
- il suffit d'assigner une valeur à un nom de variable pour que celle-ci soit automatiquement créée avec le type qui correspond au mieux à la valeur fournie.
- Par exemple :

```
n = 10      msg = "Bonjour"      euro = 6,55957
```

Python typerait automatiquement ces trois variables :

- n sera de type entier (integer)
- msg sera de type chaîne de caractères (string)
- euro sera de type réel (float)

L'instruction `type(variable)` permet de connaître le type d'une variable

```
print type(n)
```

Opérateurs de comparaison

Les opérateurs de comparaison servent à comparer des valeurs de même type et renvoient un résultat de type booléen.

Sous Python, ces opérateurs sont `<`, `<=`, `>`, `>=`, `!=`, `==`

ex. `a = (12 == 13)` # `a` est de type bool, il a la valeur False

N.B. On utilisera principalement ces opérateurs dans les branchements conditionnels.

Les expressions booléennes

Deux valeurs possibles : False, True.

Opérateurs de comparaison : ==, !=, >, >=, <, <=

```
2 > 8 # False
```

```
2 <= 8 < 15 # True
```

Opérateurs logiques : not, or, and

```
(3 == 3) or (9 > 24) # True (dès le premier membre)
```

```
(9 > 24) and (3 == 3) # False (dès le premier membre)
```

Le type entier

Un type caractérise

- la place mémoire nécessaire pour mémoriser les éléments de ce type
- les opérations qu'on peut faire sur les éléments de ce type

Opérations arithmétiques

```
20 + 3      # 23
20 - 3      # 17
20 * 3      # 60
20 ** 3     # 8000
20 / 3      # 6 (division entière)
20 % 3      # 2 (modulo)
```

Les entiers longs (seulement limités par la RAM)

```
2 ** 40      # 1099511627776L
3 * 72L      # 216L
```

Le type entier fait partie des types dits "simples"

Le type flottant

Le type flottant est la seule façon de mémoriser les nombres décimaux et les nombres réels

Les flottants sont notés avec un « point décimal »
ou en notation exponentielle :

2.718

Ils supportent les mêmes opérations que les entiers, sauf :

`20.0 / 3` # 6.666666666666667

`20.0 // 3` # 6 (division entière forcée)

Le type flottant fait partie des types dits “simples”

Priorité des opérations

PEMDAS pour le mémoriser ! ☐

1. **P** pour **parenthèses**. Ce sont elles qui ont la plus haute priorité. Elles vous permettent donc de « forcer » l'évaluation d'une expression dans l'ordre que vous voulez. Ainsi $2*(3-1) = 4$, et $(1+1)**(5-2) = 8$.
2. **E** pour **exposants**. Les exposants sont évalués avant les autres opérations. Ainsi $2**1+1 = 3$ (et non 4), et $3*1**10 = 3$ (et non 59049 !).
3. **M** et **D** pour **multiplication** et **division**, qui ont la même priorité. Elles sont évaluées avant **l'addition A** et la **soustraction S** lesquelles sont donc effectuées en dernier lieu. Ainsi $2*3-1 = 5$ (plutôt que 4), et $2/3-1 = -1$ (Rappelez-vous que par défaut Python effectue une division **entière**).
4. Si deux opérateurs ont la même priorité, l'évaluation est effectuée de gauche à droite. Ainsi dans l'expression $59*100/60$, la multiplication est effectuée en premier, et la machine doit donc ensuite effectuer $5900/60$, ce qui donne **98**. Si la division était effectuée en premier, le résultat serait **59** (rappelez-vous ici

Transtypage

Principe

Utilisation du mot-clé désignant le type

> **nouveau_type** (objet)

Conversion en numérique

a = « 12 » # a est de type chaîne caractère

b = float(a) # b est de type float

N.B. Si la conversion n'est pas possible ex. float(« toto »), Python renvoie une erreur

Conversion en logique

a = bool(« TRUE ») # a est de type bool est contient la valeur True

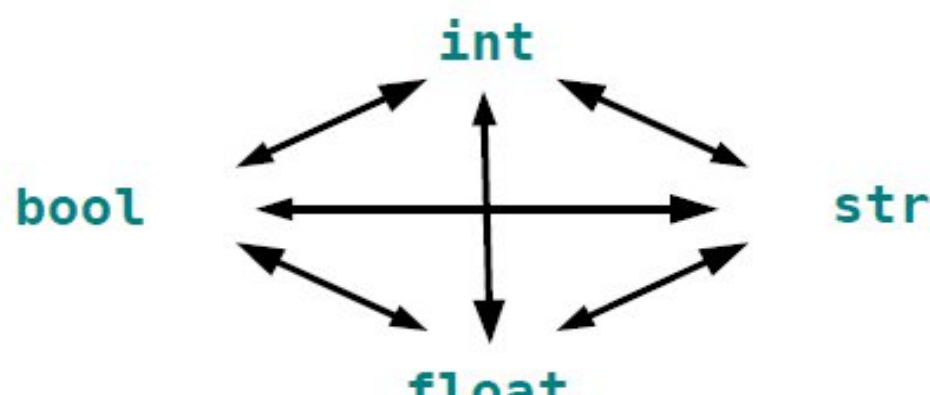
a = bool(1) # renvoie True également

Conversion en chaîne de caractères

a = str(15) # a est de type chaîne et contient « 15 »

Transtypage

Conversion de données entre les différents types



`"3" * 5`



`"33333"`

`int(11.8)`



`11`

`int("3") * 5`



`15`

`float("1.602176")`



`1.602176`

`str(3.14) + " radians"`



`"3.14 radians"`

Chapitre 2:

Entrée/Sortie et structures algorithmiques

Entrée/Sortie

On a généralement besoin de pouvoir **interagir** avec un programme :

- pour lui fournir les données à traiter, par exemple au clavier : **entrées**
- pour pouvoir connaître le résultat d'exécution pour que le programme puisse écrire ce qu'il attend de l'utilisateur,
par exemple, texte écrit à l'écran : **sorties**

Les entrées : fonction **input()**

- ❑ A l'exécution, l'ordinateur :
 - interrompt l'exécution du programme
 - affiche éventuellement un message a l'écran
 - attend que l'utilisateur entre une donnée au clavier et appuie Entrée.
- ❑ C'est une saisie en **mode texte**
 - valeur saisie vue comme une **chaîne de caractères**
 - on peut ensuite changer le type

Les entrées

```
>>> texte = input()
```

```
123
```

```
>>> texte + 1 # provoque une erreur
```

```
>>> val = int(texte)
```

```
>>> val + 1 # ok
```

```
124
```

```
>>> x = float(input("Entrez un nombre :"))
```

```
Entrez un nombre :
```

```
12.3
```

```
>>> x + 2
```

```
14.3
```

Les entrées

```
>>> texte = input()
```

```
123
```

```
>>> texte + 1 # provoque une erreur
```

```
>>> val = int(texte)
```

```
>>> val + 1 # ok
```

```
124
```

```
>>> x = float(input("Entrez un nombre :"))
```

```
Entrez un nombre :
```

```
12.3
```

```
>>> x + 2
```

```
14.3
```


Les sorties : la fonction **print()**

- ❑ Affiche la **représentation textuelle** de n'importe quel nombre de valeurs fournies entre les parenthèses et séparées par des virgules
- ❑ A l'affichage, ces valeurs sont séparées par un **espace**
- ❑ L'ensemble se termine par un retour a la ligne
 - modifiable en utilisant `sep` et/ou `end`
- ❑ Possibilité d'insérer
 - des **sauts de ligne** en utilisant `\n` et
 - des **tabulations** avec `\t`

Les sorties : la fonction **print()**

Séquence	Signification
<code>\saut_ligne</code>	saut de ligne ignoré
<code>\\</code>	affiche un antislash
<code>\'</code>	apostrophe
<code>\"</code>	guillemet
<code>\a</code>	sonnerie (<i>bip</i>)
<code>\b</code>	retour arrière
<code>\f</code>	saut de page
<code>\n</code>	saut de ligne
<code>\r</code>	retour en début de ligne
<code>\t</code>	tabulation horizontale
<code>\v</code>	tabulation verticale

Les sorties : la fonction **print()**

- ❑ La fonction **print()** affiche une chaîne de caractères .
- ❑ Elle affiche l'argument qu'on lui passe entre parenthèses et un retour à ligne
- ❑ !!retour à ligne supplémentaire est ajouté par défaut.
- ❑ Si on ne veut pas afficher ce retour à la ligne, on peut utiliser l'argument par « mot-clé » end :

```
1 >>> print("Hello world!")  
2 Hello world!  
3 >>> print("Hello world!", end="")  
4 Hello world!>>>
```

Les sorties : la fonction **print()**

Une autre manière de s'en rendre compte est d'utiliser deux fonctions `print()` à la suite.

Dans le code suivant, le caractère « ; » sert à séparer plusieurs instructions Python sur une même ligne :

```
1 >>> print("Hello") ; print("Joe")
2 Hello
3 Joe
4 >>> print("Hello", end="") ; print("Joe")
5 HelloJoe
6 >>> print("Hello", end=" ") ; print("Joe")
7 Hello Joe
```

Les sorties : la fonction **print()**

La fonction `print()` peut également afficher le contenu d'une variable quel que soit son type. Par exemple, pour un entier :

```
1 >>> var = 3
2 >>> print(var)
3 3
```

Il est également possible d'afficher le contenu de plusieurs variables (quel que soit leur type) en les séparant par des virgules :

```
1 >>> x = 32
2 >>> nom = "John"
3 >>> print(nom, "a", x, "ans")
4 John a 32 ans
```


Les sorties : la fonction **print()**

- ❑ Python a écrit une phrase complète en remplaçant les variables `x` et `nom` par leur contenu.
- ❑ Pour afficher plusieurs éléments de texte sur une seule ligne, nous avons utilisé le séparateur « , » entre les différents éléments. Python a également ajouté un espace à chaque fois que l'on utilisait le séparateur « , ».
- ❑ On peut modifier ce comportement en passant à la fonction `print()` l'argument par mot-clé `sep` :

```
1 >>> x = 32
2 >>> nom = "John"
3 >>> print(nom, "a", x, "ans", sep="")
4 Johna32ans
5 >>> print(nom, "a", x, "ans", sep="-")
6 John-a-32-ans
```

Les sorties : la fonction **print()**

Pour afficher deux chaînes de caractères l'une à côté de l'autre, sans espace, on peut soit les concaténer, soit utiliser l'argument par mot-clé `sep` avec une chaîne de caractères vide :

```
1 >>> ani1 = "chat"
2 >>> ani2 = "souris"
3 >>> print(ani1, ani2)
4 chat souris
5 >>> print(ani1 + ani2)
6 chatsouris
7 >>> print(ani1, ani2, sep="")
8 chatsouris
```

Les sorties : la fonction **print()**

❑ La méthode `.format()` permet une meilleure organisation de l'affichage des variables. Si on reprend l'exemple précédent :

```
1 >>> x = 32
2 >>> nom = "John"
3 >>> print("{} a {} ans".format(nom, x))
4 John a 32 ans
```

❑ Dans la chaîne de caractères, les accolades vides `{}` précisent l'endroit où le contenu de la variable doit être inséré.

❑ Juste après la chaîne de caractères, l'instruction `.format(nom, x)` fournit la liste des variables à insérer, d'abord la variable `nom` puis la variable `x`.

Les sorties : la fonction **print()**

- ❑ Remarque Il est possible d'indiquer entre les accolades {} dans quel ordre afficher les variables, avec 0 pour la variable à afficher en premier, 1 pour la variable à afficher en second, etc.
- ❑ (**Attention**, Python commence à compter à 0). Cela permet de modifier l'ordre dans lequel sont affichées les variables.

```
1 >>> x = 32
2 >>> nom = "John"
3 >>> print("{0} a {1} ans".format(nom, x))
4 John a 32 ans
5 >>> print("{1} a {0} ans".format(nom, x))
6 32 a John ans
```

Les sorties : la fonction `print()`

□ Imaginez maintenant que vous vouliez calculer, puis afficher, la proportion de GC d'un génome :

- La proportion de GC s'obtient comme la somme des bases Guanine (G) et Cytosine (C) divisée par le nombre total de bases (A, T, C, G) du génome considéré.
- Si on a, par exemple, 4500 bases G et 2575 bases C, pour un total de 14800 bases, vous pourriez procéder comme suit (notez bien l'utilisation des parenthèses pour gérer les priorités des opérateurs) :

```
1 >>> prop_GC = (4500 + 2575) / 14800
2 >>> print("La proportion de GC est", prop_GC)
3 La proportion de GC est 0.4780405405405405
```


Les sorties : la fonction **print()**

❑ Le résultat obtenu présente trop de décimales (seize dans le cas présent). Pour écrire le résultat plus lisiblement, vous pouvez spécifier dans les accolades {} le format qui vous intéresse.

- Exemple :

```
1 >>> print("La proportion de GC est {:.2f}".format(prop_GC))  
2 Le proportion de GC est 0.48  
3 >>> print("La proportion de GC est {:.3f}".format(prop_GC))  
4 La proportion de GC est 0.478
```

- Détaillons le contenu des accolades de la première ligne ({:.2f}) :

- Les deux points : indiquent qu'on veut préciser le format.
- La lettre f indique qu'on souhaite afficher la variable sous forme d'un float.
- Les caractères .2 indiquent la précision voulue, soit ici deux chiffres après la virgule.

Les sorties : la fonction **print()**

❑ Notez enfin que le formatage avec `.xf` (x étant un entier positif) renvoie un résultat arrondi. Il est par ailleurs possible de combiner le formatage (à droite des 2 points) ainsi que l'emplacement des variables à substituer (à gauche des 2 points), par exemple :

```
1 >>> print("prop GC(2 déci.) = {0:.2f}, prop GC(3 déci.) = {0:.3f}".format(prop_GC))
2 prop GC(2 déci.) = 0.48, prop GC(3 déci.) = 0.478
```

- remarque : la même variable (`prop_GC`) à deux endroits différents.

❑ Vous pouvez aussi formater des entiers avec la lettre `d`

```
1 >>> nb_G = 4500
2 >>> print("Ce génome contient {:d} guanines".format(nb_G))
3 Ce génome contient 4500 guanines
```

Les sorties : la fonction **print()**

- ❑ On peut mettre plusieurs nombres dans une même chaîne de caractères

```
1 >>> nb_G = 4500
2 >>> nb_C = 2575
3 >>> print("Ce génome contient {:d} G et {:d} C, soit une prop de GC de {:.2f}" \
4 ... .format(nb_G,nb_C,prop_GC))
5 Ce génome contient 4500 G et 2575 C, soit une prop de GC de 0.48
6 >>> perc_GC = prop_GC * 100
7 >>> print "Ce génome contient {:d} G et {:d} C, soit un %GC de {:.2f} %" \
8 ... .format(nb_G,nb_C,perc_GC)
9 Ce génome contient 4500 G et 2575 C, soit un %GC de 47.80 %
```

- ❑ **Remarque** : Le signe `\` en fin de ligne permet de poursuivre la commande sur la ligne suivante. Cette syntaxe est pratique lorsque vous voulez taper une commande longue.

Les sorties : la fonction **print()**

❑ Il est possible de préciser sur combien de caractères vous voulez qu'un résultat soit écrit et comment se fait l'alignement (à gauche, à droite ou centré).

- Exemple : le caractère **;** sert de séparateur entre les instructions sur une même ligne :

```
1 >>> print(10) ; print(1000)
2 10
3 1000
4 >>> print("{:>6d}".format(10)) ; print("{:>6d}".format(1000))
5      10
6      1000
7 >>> print("{:<6d}".format(10)) ; print("{:<6d}".format(1000))
8 10
9 1000
10 >>> print("{:^6d}".format(10)) ; print("{:^6d}".format(1000))
11      10
12      1000
13 >>> print("{:*^6d}".format(10)) ; print("{:*^6d}".format(1000))
14 **10**
15 *1000*
16 >>> print("{:0>6d}".format(10)) ; print("{:0>6d}".format(1000))
17 000010
18 001000
```


Les sorties : la fonction **print()**

❑ Notez que:

- > spécifie un alignement à droite,
- < spécifie un alignement à gauche
- ^ spécifie un alignement centré.

❑ Il est également possible d'indiquer le caractère qui servira de remplissage lors des alignements (l'espace est le caractère par défaut).

```
1 >>> print("atom HN") ; print("atom HDE1")
2 atom HN
3 atom HDE1
4 >>> print("atom {:>4s}".format("HN")) ; print("atom {:>4s}".format("HDE1"))
5 atom   HN
6 atom HDE1
```


Les sorties : la fonction **print()**

- ❑ L'avantage de l'écriture formatée. Elle vous permet d'écrire en colonnes parfaitement alignées
- ❑ Pour les floats, il est possible de combiner le nombre de caractères à afficher avec le nombre de décimales

```
1 >>> print("{:7.3f}".format(perc_GC))
2 47.804
3 >>> print("{:10.3f}".format(perc_GC))
4 47.804
```

Les sorties : la fonction **print()**

- ❑ L'instruction `7.3f` signifie que l'on souhaite écrire un float avec 3 décimales et formaté sur 7 caractères (par défaut justifiés à droite).
- ❑ L'instruction `10.3f` fait la même chose sur 10 caractères.
- ❑ Remarquez que le séparateur décimal `.` compte pour un caractère.
- ❑ Si on veut afficher des accolades littérales et utiliser la méthode `.format()` en même temps, il faut doubler les accolades pour échapper au formatage.

```
1 >>> print("Accolades littérales {}{} et pour le formatage {}".format(10))  
2 Accolades littérales {} et pour le formatage 10
```

Les sorties : la fonction **print()**

❑ La méthode `.format()` agit sur la chaîne de caractères à laquelle elle est attachée par un point et n'a rien à voir avec la fonction `print()`.

❑ Si on donne une chaîne de caractères suivie d'un `.format()` à la fonction `print()`, Python évalue d'abord le formatage et c'est la chaîne de caractères qui en résulte qui est affichée à l'écran.

Exemple : Tout comme dans l'instruction `print(5*5)`, c'est d'abord la multiplication $(5*5)$ qui est évaluée, puis son résultat qui est affiché à l'écran.

```
1 >>> "{:10.3f}".format(perc_GC)
2 '      47.804'
3 >>> type("{:10.3f}".format(perc_GC))
4 <class 'str'>
```

Python affiche le résultat de l'instruction `"{:10.3f}".format(perc_GC)` comme une chaîne de caractères et la fonction `type()` nous le confirme.

Les sorties : la fonction **print()**

Ancienne méthode de formatage des chaînes de caractère

❑ Dans d'anciens livres ou programmes Python, il se peut que vous rencontriez l'écriture formatée avec le style suivant :

```
1 >>> x = 32
2 >>> nom = "John"
3 >>> print("%s a %d ans" % (nom, x))
4 John a 32 ans
5 >>> nb_G = 4500
6 >>> nb_C = 2575
7 >>> prop_GC = (nb_G + nb_C)/14800
8 >>> print("On a %d G et %d C -> prop GC = %.2f" % (nb_G, nb_C, prop_GC))
9 On a 4500 G et 2575 C -> prop GC = 0.48
```

Les sorties : la fonction **print()**

❑ La syntaxe est légèrement différente. Le symbole % est d'abord appelé dans la chaîne de caractères (dans l'exemple ci dessus %d, %d et %.2f) pour :

- Désigner l'endroit où sera placée la variable dans la chaîne de caractères.
- Préciser le type de variable à formater, d pour un entier (i fonctionne également) ou f pour un float.

Les sorties : la fonction **print()**

Note sur le vocabulaire et la syntaxe

- ❑ En Python, on peut considérer chaque variable comme un objet sur lequel on peut appliquer des méthodes.
- ❑ Une méthode est simplement une fonction qui utilise et/ou agit sur l'objet lui-même, les deux étant connectés par un point. La syntaxe générale est de la forme **objet.méthode()**.

```
1 >>> "Joe a {} ans".format(20)  
2 'Joe a 20 ans'
```

- ❑ la méthode `.format()` est liée à `"Joe a {} ans"` qui est un objet de type chaîne de caractères. La méthode renvoie une nouvelle chaîne de caractères avec le bon formatage (ici, `'Joe a 20 ans'`).

Chapitre 3:

structures algorithmiques :
branchements conditionnels
et boucles

Branchement conditionnel « if »

❑ **Objectif : effectuer des actions seulement si** une certaine condition est vérifiée

❑ **Syntaxe en Python :**

if condition :

instructions à exécuter si vrai

La condition est une **expression booléenne**

- **Attention à l'indentation !**

- Indique dans quel bloc se trouve une instruction.
- obligatoire en Python.

Branchement conditionnel « if »

Condition est très souvent une opération de comparaison



```
if condition:  
    bloc d'instructions  
else:  
    bloc d'instructions
```

- (1) Attention au **:** qui est primordial
- (2) C'est l'**indentation** (le décalage par rapport à la marge gauche) qui délimite le bloc d'instructions
- (3) La partie **else** est facultative

Branchement conditionnel « if »: exemple

Noter l'imbrication des blocs.

Le code appartenant au même bloc doit être impérativement aligné sinon erreur.

```
calc_tva_conditionnel.py - D:\Travaux\university\Cours_Univ...
File Edit Format Run Options Window Help
# -*- coding: utf -*-

#saisie à la console d'un prix HT
pht = float(input("prix HT : "))

#code produit
code = int(input("code de produit : "))

#action conditionnelle
if (code == 1):
    taxe = pht * 0.055
    pttc = pht + taxe
else:
    pttc = pht * 1.2

#affichage avec transtypage
print("prix TTC : " + str(pttc))

#pour bloquer la fermeture de la console
input("pause...")
```

Ln: 1 Col: 0

Succession de if avec elif

```
calc_tva_elif.py - D:\Travaux\university...
File Edit Format Run Options Window Help
# -*- coding: utf -*-

#saisie à la console d'un prix HT
pht = float(input("prix HT : "))

#code produit
code = int(input("code de produit : "))

#action conditionnelle
if (code == 1):
    pttc = pht * 1.055
elif (code == 2):
    pttc = pht * 1.1
else:
    pttc = pht * 1.2

#affichage avec transtypage
print("prix TTC : " + str(pttc))

#pour bloquer la fermeture de la console
input("pause...")

Ln: 1 Col: 0
```

- **elif** n'est déclenché que si la (les) condition(s) précédente(s) a (ont) échoué.
- **elif** est situé au même niveau que **if** et **else**
- On peut en mettre autant que l'on veut



Il n'y a pas de `switch()` ou de `case...of` en Python

Avant la boucle « for »: génération d'une séquence de valeurs

Principe de la boucle for

Elle ne s'applique que sur une collection de valeurs. Ex. tuples, listes,... à voir plus tard.

Suite arithmétique simple (séquence de valeurs entières)

On peut définir des boucles indicées en générant une collection de valeurs avec `range()`


(1) `range(4)` → 0 1 2 3

(2) `range(1, 4)` → 1 2 3

(3) `range(0, 5, 2)` → 0 2 4

Boucle « for »

Séquence est une collection de valeurs
Peut être générée avec range()



```
for indice in séquence:  
    bloc d'instructions
```

Remarque :

- Attention à l'indentation toujours
- On peut « casser » la boucle avec **break**
- On peut passer directement à l'itération suivante avec **continue**
- Des boucles imbriquées sont possibles
- Le bloc d'instructions peut contenir des conditions

Boucle « for »: exemple

Somme totale des valeurs comprises entre 1 et **n** (inclus) et somme des valeurs paires dans le même intervalle

```
somme_paire.py - D:\Travaux\university\Cours_Univer... - [X]
File Edit Format Run Options Window Help
# -*- coding: utf -*-
#valeur limite
n = int(input("n : "))
#initialisation
sommePaire = 0
sommeTotale = 0
#effectuer la somme
for i in range(1,n+1):
    #somme si valeur paire
    if (i % 2 == 0):
        sommePaire = sommePaire + i
    #on n'est plus dans le " if " ici
    #mais on est bien dans le " for "
    sommeTotale = sommeTotale + i
#on est sorti du " for " ici
#affichage avec transtypage
print("somme des valeurs paires : " + str(sommePaire))
print("somme totale : " + str(sommeTotale))
#pour bloquer la fermeture de la console
input("pause...")
```

Il faut mettre **n+1** dans `range()` pour que **n** soit inclus dans la somme

Observez attentivement les indentations.

Boucle « while » (exemple)

Opération de comparaison
Attention à la boucle infinie !



```
while condition:  
    bloc d'instructions
```

Remarque :

- Attention à l'indentation toujours
- On peut « casser » la boucle avec **break**

Boucle « while » (exemple)

```
somme_paire_while.py - D/_Travaux/university/Co... - □ ×
File Edit Format Run Options Window Help
# -*- coding: utf -*-

#valeur limite
n = int(input("n : "))

#initialisation
sommePaire = 0
sommeTotale = 0

#effectuer la somme
i = 1
while (i <= n):
    #somme si valeur paire
    if (i % 2 == 0):
        sommePaire = sommePaire + i
    #somme toujours, paire ou pas
    sommeTotale = sommeTotale + i
    #incréméntation
    i = i + 1

#affichage avec transtypage
print("somme des valeurs paires : " + str(sommePaire))
print("somme totale : " + str(sommeTotale))

#pour bloquer la fermeture de la console
input("pause...")
|
```

Ne pas oublier
l'initialisation de **i**

Observez attentivement
les indentations.

Ln: 28 Col: 0



Boucle « while » (exemple)

Exemple: division de A par B

Un programme qui demande un entier A, puis un entier B jusqu'à ce que celui-ci soit non-nul, puis qui calcule le quotient de A par B.

```
a = int(input("Donnez la valeur de A : "))  
b = int(input("Donnez la valeur de B : "))  
while b == 0 :  
    b = int(input("B est nul! Recommencez : "))  
print("A / B = ", a // b)
```

*Note: si b est non-nul dès le premier essai, on n'entre **pas** dans le while.*

Boucle « while » (exemple)

Attention aux boucles infinies!

Si la condition de la boucle while ne devient jamais fausse, le programme boucle **indéfiniment**:

Exemple 1:

```
n=5
```

```
while n<10 :
```

```
    print("n vaut :", n)
```

```
    print("Fin")
```

Exemple 2:

```
while True :
```

```
    print("Je boucle.")
```

```
    print("Fin")
```

Boucle « while » (exemple)

Le mot-cle **break**

- Permet de **sortir immédiatement de la boucle**

wh

```
i=1
```

```
while i<100:
```

```
    if i % 2 == 0 :
```

```
        print("*")
```

```
        break
```

```
    i=i+1
```

```
    print("Incrémentation de i")
```

```
print("Fin")
```

Boucle « while » (exemple)

Le mot-clé : **continue**

- ❑ Permet de **remonter immédiatement au début de la boucle**

while en ignorant la suite des instructions

```
for i in range(4):  
    print("debut iteration", i)  
    print("bonjour")  
    if i < 2:  
        continue  
    print("fin iteration", i)  
print("après la boucle")
```



```
debut iteration 0  
bonjour  
debut iteration 1  
bonjour  
debut iteration 2  
bonjour  
fin iteration 2  
debut iteration 3  
bonjour  
fin iteration 3  
apres la boucle
```


Break et continue

Inconvénients:

- ❑ Code plus difficile à lire/analyser si plusieurs niveaux d'imbrications et/ou longues instructions dans le while
- ❑ N'a pas toujours d'équivalent dans les autres langages de programmation

➡ On essaiera tant que possible de se passer de break et continue.

Boucle « while » (exemple)

Boucles imbriquées

- Une instruction d'une boucle while peut être une boucle while

Ex : résultat produit par ce programme ?

```
i = 1
```

```
while i <= 3 :
```

```
    j = 1
```

```
    while j <= 2 :
```

```
        print(i, ", ", j)
```

```
        j = j + 1
```

```
    i = i + 1
```

Boucle « while »

Boucles imbriquées

On veut écrire un programme qui affiche un « carre » de $n \times n$

fois le caractère '*'. L'utilisateur choisit le cote n du carre.

Exemple de résultat :

Entrez la valeur de n : 5

Boucle « while »

Correction

```
n = int(input("Entrez la valeur de n : "))
num_lig = 0 # compteur de ligne
while num_lig < n :
    num_col = 0 #cpt nb etoiles de la ligne
    while num_col < n :
        print("*", end="") # /\ end
        num_col = num_col + 1
    print() # saut de ligne
    num_lig = num_lig + 1 # passer ligne suivante
```

Chapitre 4:

Les listes

Les listes

Utilisation

- ❑ Une liste est une structure de données qui contient une série de valeurs.
- ❑ Python autorise la construction de liste contenant des valeurs de types différents (par exemple entier et chaîne de caractères).
- ❑ Une liste est déclarée par une série de valeurs (n'oubliez pas les guillemets, simples ou doubles, s'il s'agit de chaînes de caractères) séparées par des virgules, et le tout encadré par des crochets. Exemples :

```
1 >>> animaux = ['girafe', 'tigre', 'singe', 'souris']
2 >>> tailles = [5, 2.5, 1.75, 0.15]
3 >>> mixte = ['girafe', 5, 'souris', 0.15]
4 >>> animaux
5 ['girafe', 'tigre', 'singe', 'souris']
6 >>> tailles
7 [5, 2.5, 1.75, 0.15]
8 >>> mixte
9 ['girafe', 5, 'souris', 0.15]
```

Les listes

Utilisation

❑ Exemples :

```
1 >>> animaux = ['girafe', 'tigre', 'singe', 'souris']
2 >>> tailles = [5, 2.5, 1.75, 0.15]
3 >>> mixte = ['girafe', 5, 'souris', 0.15]
4 >>> animaux
5 ['girafe', 'tigre', 'singe', 'souris']
6 >>> tailles
7 [5, 2.5, 1.75, 0.15]
8 >>> mixte
9 ['girafe', 5, 'souris', 0.15]
```

Les listes

Utilisation

❑ Un des gros avantages d'une liste est que vous pouvez appeler ses éléments par leur position. Ce numéro est appelé indice(ou index) de la liste.

```
1 liste : ['girafe', 'tigre', 'singe', 'souris']  
2 indice :      0      1      2      3
```

❑ Soyez très attentifs au fait que les indices d'une liste de n éléments commence à 0 et se termine à $n-1$. Voyez l'exemple suivant :

```
1 >>> animaux = ['girafe', 'tigre', 'singe', 'souris']  
2 >>> animaux[0]  
3 'girafe'  
4 >>> animaux[1]  
5 'tigre'  
6 >>> animaux[3]  
7 'souris'
```

Les listes

Utilisation

❑ Si on appelle l'élément d'indice 4 de notre liste, Python renverra un message d'erreur :

```
1 liste : ['girafe', 'tigre', 'singe', 'souris']  
2 indice :      0      1      2      3
```

```
1 >>> animaux[4]  
2 Traceback (innermost last):  
3   File "<stdin>", line 1, in ?  
4 IndexError: list index out of range
```

Les listes

Opération sur les listes

- ❑ Tout comme les chaînes de caractères, les listes supportent l'opérateur + de concaténation, ainsi que l'opérateur * pour la duplication :

```
1 >>> ani1 = ['girafe', 'tigre']  
2 >>> ani2 = ['singe', 'souris']  
3 >>> ani1 + ani2  
4 ['girafe', 'tigre', 'singe', 'souris']  
5 >>> ani1 * 3  
6 ['girafe', 'tigre', 'girafe', 'tigre', 'girafe', 'tigre']
```

- ❑ L'opérateur + est très pratique pour concaténer deux listes.
- ❑ Vous pouvez aussi utiliser la méthode **.append()** lorsque vous souhaitez ajouter un seul élément à la fin d'une liste.

Les listes

Opération sur les listes

❑ Exemple de création d'une liste vide

```
1 >>> a = []  
2 >>> a  
3 []
```

❑ Puis lui ajouter deux éléments, l'un après l'autre, d'abord avec la concaténation :

```
1 >>> a = a + [15]  
2 >>> a  
3 [15]  
4 >>> a = a + [-5]  
5 >>> a  
6 [15, -5]
```

❑ Puis avec la méthode .append() :

```
1 >>> a.append(13)  
2 >>> a  
3 [15, -5, 13]  
4 >>> a.append(-3)  
5 >>> a  
6 [15, -5, 13, -3]
```

Les listes

Opération sur les listes

- ❑ Dans l'exemple ci-dessus, nous ajoutons des éléments à une liste en utilisant l'opérateur de concaténation + ou la méthode `.append()`.
- ❑ Nous vous conseillons dans ce cas précis d'utiliser la méthode `.append()` dont la syntaxe est plus élégante.
- ❑ Nous reverrons en détail la méthode `.append()` dans la suite du cours

Les listes

Indexage négatif

- ❑ La liste peut également être indexée avec des nombres négatifs selon le modèle suivant :

```
1 liste          : ['girafe', 'tigre', 'singe', 'souris']
2 indice positif :      0         1         2         3
3 indice négatif :     -4        -3        -2        -1
```

ou encore :

```
1 liste          : ['A', 'B', 'C', 'D', 'E', 'F']
2 indice positif :      0      1      2      3      4      5
3 indice négatif :     -6     -5     -4     -3     -2     -1
```

- ❑ Les indices négatifs reviennent à compter à partir de la fin.
- ❑ Leur principal avantage est que vous pouvez accéder au dernier élément d'une liste à l'aide de l'indice -1 sans pour autant connaître la longueur de cette liste.
- ❑ L'avant-dernier élément a lui l'indice -2, l'avant-avant dernier l'indice -3, etc.

Les listes

Indexage négatif

```
1 >>> animaux = ['girafe', 'tigre', 'singe', 'souris']
2 >>> animaux[-1]
3 'souris'
4 >>> animaux[-2]
5 'singe'
```

❑ Pour accéder au premier élément de la liste avec un indice négatif, il faut par contre connaître le bon indice :

```
1 >>> animaux[-4]
2 'girafe'
```

❑ Dans ce cas, on utilise plutôt animaux[0].

Les listes

Tranche

❑ Un autre avantage des listes est la possibilité de sélectionner une partie d'une liste en utilisant **un indexage** construit sur le **modèle $[m:n+1]$** pour récupérer tous les éléments, du m ème au n ème (de l'élément m inclus à l'élément $n+1$ exclu).

❑ On dit alors qu'on récupère une tranche de la liste, par exemple

```
1 animaux = ['girafe', 'tigre', 'singe', 'souris']
2 >>> animaux[0:2]
3 ['girafe', 'tigre']
4 >>> animaux[0:3]
5 ['girafe', 'tigre', 'singe']
6 >>> animaux[0:]
7 ['girafe', 'tigre', 'singe', 'souris']
8 >>> animaux[:]
9 ['girafe', 'tigre', 'singe', 'souris']
10 >>> animaux[1:]
11 ['tigre', 'singe', 'souris']
12 >>> animaux[1:-1]
13 ['tigre', 'singe']
```


Les listes

Tranche

- ❑ Notez que lorsqu'aucun indice n'est indiqué à gauche ou à droite du symbole deux-points.
- ❑ Python prend par défaut tous les éléments depuis le début ou tous les éléments jusqu'à la fin respectivement.
- ❑ On peut aussi préciser le pas en ajoutant un symbole deux-points supplémentaire et en indiquant le pas par un entier.

```
animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> animaux[0:3:2]
['girafe', 'singe']
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::1]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::2]
[0, 2, 4, 6, 8]
>>> x[::3]
[0, 3, 6, 9]
>>> x[1:6:3]
[1, 4]
```

l'accès au contenu d'une liste
fonctionne sur le modèle:
liste[début:fin:pas]

Les listes

Fonction: **len()**

❑ L'instruction **len()** vous permet de connaître la longueur d'une liste, c'est-à-dire le nombre d'éléments que contient la liste. Voici un exemple d'utilisation :

```
1 >>> animaux = ['girafe', 'tigre', 'singe', 'souris']
2 >>> len(animaux)
3 4
4 >>> len([1, 2, 3, 4, 5, 6, 7, 8])
5 8
```

Les listes

Les fonctions: **range()** et **list()**

- ❑ L'instruction **range()** est une fonction spéciale en Python qui génère des nombres entiers compris dans un intervalle.
- ❑ Lorsqu'elle est utilisée en combinaison avec la fonction **list()**, on obtient une liste d'entiers. Par exemple :

```
1 >>> list(range(10))  
2 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

La commande `list(range(10))` a généré une liste contenant tous les nombres entiers de 0 inclus à 10 exclu. Nous verrons l'utilisation de la fonction `range()` toute seule dans le chapitre 5 Boucles et comparaisons.

Dans l'exemple ci-dessus, la fonction `range()` a pris un argument, mais elle peut également prendre deux ou trois arguments,

VOYONS SILEZ!

```
1 >>> list(range(0, 5))  
2 [0, 1, 2, 3, 4]  
3 >>> list(range(15, 20))  
4 [15, 16, 17, 18, 19]  
5 >>> list(range(0, 1000, 200))  
6 [0, 200, 400, 600, 800]  
7 >>> list(range(2, -2, -1))  
8 [2, 1, 0, -1]
```

Les listes

Les fonctions: **range()** et **list()**

❑ Dans l'exemple ci-dessus, la fonction `range()` a pris un argument, mais elle peut également prendre deux ou trois arguments:

```
1 >>> list(range(0, 5))
2 [0, 1, 2, 3, 4]
3 >>> list(range(15, 20))
4 [15, 16, 17, 18, 19]
5 >>> list(range(0, 1000, 200))
6 [0, 200, 400, 600, 800]
7 >>> list(range(2, -2, -1))
8 [2, 1, 0, -1]
```

Les listes

Les fonctions: **range()** et **list()**

- ❑ L'instruction `range()` fonctionne sur le modèle `range([début,] fin[, pas])`.
- ❑ Les arguments entre crochets sont optionnels. Pour obtenir une liste de nombres entiers, il faut l'utiliser systématiquement avec la fonction `list()`.
- ❑ Prenez garde aux arguments optionnels par défaut (0 pour début et 1 pour pas) :

```
1 | >>> list(range(10,0))  
2 | []
```

Les listes

Les fonctions: **range()** et **list()**

- ❑ Ici la liste est vide car Python a pris la valeur du pas par défaut qui est de 1.
- ❑ Ainsi, si on commence à 10 et qu'on avance par pas de 1, on ne pourra jamais atteindre 0.
- ❑ Python génère ainsi une liste vide. Pour éviter ça, il faudrait, par exemple, préciser un pas de -1 pour obtenir une liste d'entiers décroissants :

```
1| >>> list(range(10,0,-1))  
2| [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```


Les listes

Listes de listes

❑ Il est tout à fait possible de construire des listes de listes. Cette fonctionnalité peut parfois être très pratique.

❑ Exemple

```
1 >>> enclos1 = ['girafe', 4]
2 >>> enclos2 = ['tigre', 2]
3 >>> enclos3 = ['singe', 5]
4 >>> zoo = [enclos1, enclos2, enclos3]
5 >>> zoo
6 [['girafe', 4], ['tigre', 2], ['singe', 5]]
```

❑ Dans cet exemple, chaque sous-liste contient une catégorie d'animal et le nombre d'animaux pour chaque catégorie.

❑ Pour accéder à un élément de la liste, on utilise l'indiciage habituel

```
>>> zoo[1]
['tigre', 2]
```

Les listes

Listes de listes

- ❑ Pour accéder à un élément de la sous-liste, on utilise un double indiciage

```
1 >>> zoo[1][0]
2 'tigre'
3 >>> zoo[1][1]
4 2
```

- ❑ On verra un peu plus loin qu'il existe en Python des **dictionnaires** qui sont également très pratiques pour stocker de l'information structurée.
- ❑ On verra aussi qu'il existe **un module** nommé **NumPy** qui permet de créer des listes ou des tableaux de nombres (vecteurs et matrices) et de les manipuler.

Les listes

❑ Fonction: **.insert()**

- La méthode **.insert()** insère un objet dans une liste avec un indice déterminé :

```
1 >>> a.insert(2, -15)
2 >>> a
3 [1, 2, -15, 3, 5]
```

❑ Fonction: **.del()**

- L'instruction **del** supprime un élément d'une liste à un indice déterminé :

```
1 >>> del a[1]
2 >>> a
3 [1, -15, 3, 5]
```

❑ **Remarque** : Contrairement aux autres méthodes associées aux listes, **del** est une instruction générale de Python, utilisable pour d'autres objets que des listes. Celle-ci ne prend pas de parenthèse.

Les listes

❑ Fonction: **.remove()**

- La méthode **.remove()** supprime un élément d'une liste à partir de sa valeur :

```
1 | >>> a.remove(5)
2 | >>> a
3 | [1, -15, 3]
```

❑ Fonction: **.sort()**

- La méthode **.sort()** trie une liste :

```
1 | >>> a.sort()
2 | >>> a
3 | [-15, 1, 3]
```

Les listes

▪ Fonction: **.reverse()**

- La méthode **.reverse()** inverse une liste :

```
1 | >>> a.reverse()  
2 | >>> a  
3 | [3, 1, -15]
```

Fonction: **.count()**

- La méthode **.count()** compte le nombre d'éléments (passés en argument) dans une liste :

```
1 | >>> a=[1, 2, 4, 3, 1, 1]  
2 | >>> a.count(1)  
3 | 3  
4 | >>> a.count(4)  
5 | 1  
6 | >>> a.count(23)  
7 | 0
```

Chapitre 5

Les tuples

Les tuples

Utilisation

- ❑ Il permet de créer une **collection ordonnée de plusieurs éléments**. En mathématiques, on parle de **p-uplet**. Par exemple, un quadruplet est constitué de 4 éléments.
- ❑ Les tuples ressemblent aux listes, mais on ne peut pas les modifier une fois qu'ils ont été créés.
- ❑ On dit qu'un tuple **n'est pas mutable**.
- ❑ On le définit avec des **parenthèses**.

Exemple:

```
>>> a=(3,4,7)
>>> type(a)
```

Les tuples

- ❑ Parfois, les tuples ne sont pas entourés de parenthèses, même s'il s'agit quand même de tuples.
- ❑ Ainsi, on peut utiliser la notation suivante :

```
>>> b, c = 5, 6
```

```
>>> b
```

```
5
```

```
>>> c
```

```
6
```

- ❑ En fait, cela revient à :

```
>>> (b, c) = (5, 6)
```

- ❑ On peut écrire aussi :

```
>>> a = (3, 4)
```

```
>>> u, v = a
```

```
>>> u
```

```
3
```

```
>>> v
```

```
4
```

Les tuples

- ❑ Comme une liste, il est possible de parcourir un tuple avec une boucle **for**
- ❑ Ainsi, on peut utiliser la syntaxe suivante :
for i **in** a:
 print (i)
- ❑ La valeur d'un élément du tuple est obtenue en utilisant la même syntaxe que pour une liste.

```
>>> a[0]
```

```
3
```

```
>>> a[1]
```

```
4
```

- ❑ Pour créer un tuple à un seul élément, il faut utiliser une syntaxe avec une virgule.

```
>>> b=(3,)
```

```
>>> type(b)
```

```
<class 'tuple'>
```

Les tuples

- ❑ Si on veut récupérer l'unique valeur présente dans le tuple, on va pouvoir utiliser l'une des approches suivantes:

- ❑ Première approche:

```
>>> c = b[0]
```

```
>>> c
```

```
3
```

- ❑ Deuxième approche:

```
>>> d, = b
```

```
>>> d
```

```
3
```

Il est possible d'utiliser la syntaxe `nom_de_variable, =` aussi avec une liste à un élément.

```
>>> u = [5]
```

```
>>> v, = u
```

```
>>> v
```

```
5
```

Chapitre 6

Les dictionnaires

Les dictionnaires

Utilisation

- ❑ Un **dictionnaire** en Python va aussi permettre de rassembler des éléments mais ceux-ci seront identifiés par une **clé**. On peut faire l'analogie avec un dictionnaire de français où on accède à une définition avec un mot.
- ❑ On le définit avec des **accolades**.

Exemple:

```
>>> mon_dictionnaire={"voiture": "véhicule à quatre roues", "velo": "véhicule à deux roues"}
```

clé: valeur

```
>>> mon_dictionnaire["voiture"]  
véhicule à quatre roues
```


Les dictionnaires

❑ Pour ajouter un élément à un dictionnaire, il suffit d'affecter une **valeur** pour la nouvelle clé.

```
>>> mon_dictionnaire ["tricycle"] = "véhicule à trois roues"
```

```
>>> mon_dictionnaire  
{'voiture': 'véhicule à quatre roues', 'vélo': 'véhicule à deux  
roues', 'tricycle': 'véhicule à trois roues' }
```

❑ Pour créer un dictionnaire, il suffit de créer un dictionnaire vide et à ajouter les éléments au fur et à mesure.

```
>>> nombre_de_pneus = {}  
>>> nombre_de_pneus["tricycle"] = 3  
>>> nombre_de_pneus["vélo"] = 2  
>>> nombre_de_pneus  
{'voiture': 4, 'vélo': 2}
```

Les dictionnaires

- ❑ Pour parcourir un dictionnaire, il suffit d'utiliser **items()**.

```
Nombre_de_roues={"voiture": 4, "vélo": 2, "tricycle":3}
```

```
for i in nombre_de_roues.items():  
    print(i)
```

output

('voiture', 4)

('vélo', 2)

('tricycle', 3)

```
for cle, valeur in nombre_de_roues.items():  
    print("l' élément de clé ", cle, "vaut", valeur)
```

output

l' élément de clé voiture vaut 4

l' élément de clé vélo vaut 2

l' élément de clé tricycle vaut 3