Formation Python

Alaa El Yassir

alaa.elyassir@gmail.com

Programme

- Présentation Python
- Les types de bases
- Les fonctions
- La programmation objet
- La gestion d'exception
- L'ouverture de fichier
- L'importation de module
- Stlib : La sérialisation, gestion bases de données, module "os" et "re"
- POO avancée
- Quality assurance
- Ihm
- Interfaçage C/Python

Partage docs :
https://drive.google.com/drive/folders/16Uhurh6zuLJm6QJ0H-3XTMIGFHX8ePDO?usp=sharing

alaa.elyassir@gmail.com

I. Introduction Python

alaa.elyassir@gmail.com

Historique

- Python est un langage de programmation créé en 1989 par Guido Van Rossum.
- En 2001 la Python Software Foundation a été créée, organisation à but non lucratif.
- Python est sous licence GPL depuis 2001.

Version	Année d'apparition	Fonctionnalités supplémentaires
Python	1989	Inspiré du langage ABC
Python 0,9	1991	Première diffusion du code Python
Python 2	2001	Support d'Unicode
Python 3	2009	Nombreuses fonctionnalités ajoutées ou modifiées. Suppression d'éléments obsolètes

Caractéristiques

Langage interprété

Python est un langage interprété, il n'a pas besoin d'être compilé.

Langage orienté objet

Les langages orientés objet tels que Python permettent de développer des applications structurées.

Cependant la programmation orientée objet est une option en Python, il n'est pas nécessaire de maîtriser la programmation objet pour développer en python.

Portabilité

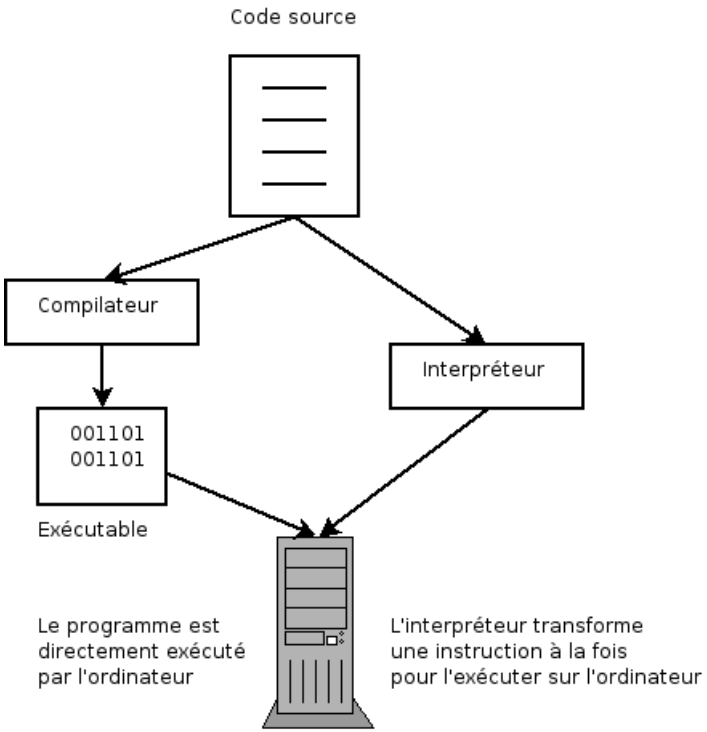
Python s'exécute sur pratiquement toutes les plateformes utilisées actuellement : Unix, Windows, MacOS, VMS, QNX, …

Python est d'ailleurs installé par défaut sur de nombreux systèmes Unix.

Puissant et diversifié

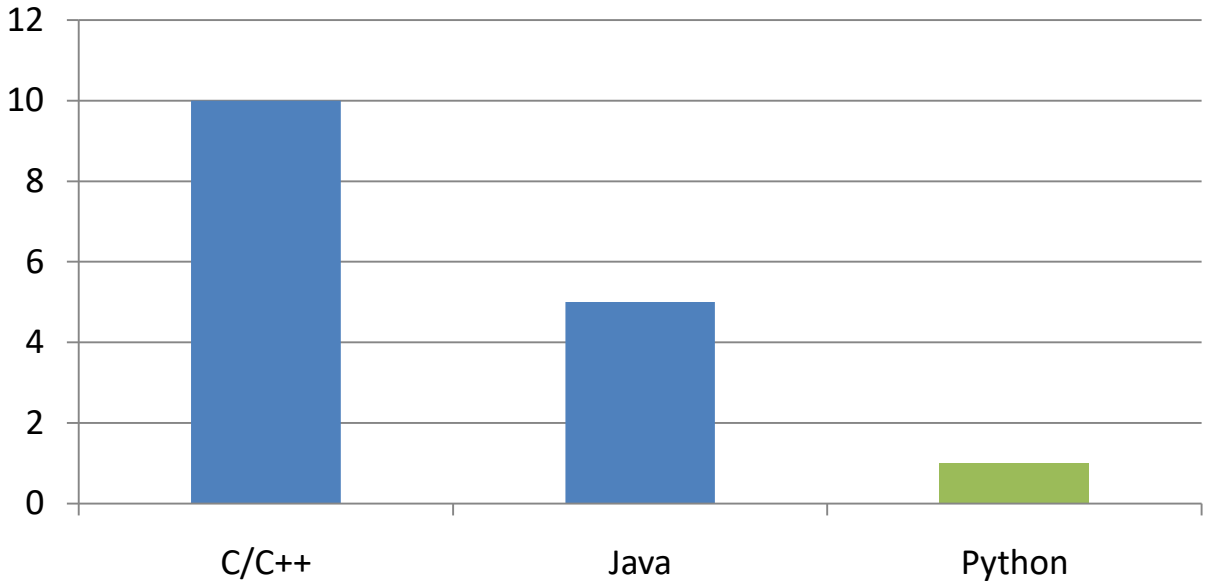
Python préserve la facilité d'utilisation des langages scripts (Shell, Tcl, ...) tout en fournissant de nombreux outils avancés typiques des langages de programmation objet (C++, Java, …).

Langage interprété



Source : http://www.ukonline.be/programmation/java/tutoriel/intro.php

Développement projet : coût



Lien avec d'autres langages

CPython

Il existe une API d'intégration Python/C qui permet d'étendre les programmes Python par des composants écrits en C.
Ceci peut permettre d'implémenter un système en Python afin de réduire le temps de développement puis, si besoin, les éléments qui le nécessitent peuvent être réécrits en C.

JPython

JPython permet la communication entre les programmes Python et des composants Java. Il peut être utile, par exemple, pour fédérer des applications web en Java.

IronPython

IronPython est l'implémentation de Python pour la plateforme .NET.

PythonWin

PythonWin a été conçu pour les plateformes MS-Windows. Il rend possible l'interaction entre les programmes Python et des composants écrits avec l'API COM.

Installation

Sous Windows

- Rendez-vous sur le site officiel de Python : http://python.org/
- Dans le menu à gauche, allez dans la rubrique 'Download'.
- Sélectionnez une version Python x.x
- Ensuite, dans le paragraphe 'Download' plusieurs formats sont proposés pour le téléchargement.
- Sélectionnez la version qui conviendra à votre processeur (Windows X86 par exemple).
- Enregistrez le fichier puis exécutez le.

Installation

Sous Linux

Python est déjà installé sur la plupart des systèmes, cependant votre version risque d'être une version plus ancienne (2.x). Vous pouvez le vérifiez avec la commande : **`python -V`**

Pour installer une nouvelle version :

- Rendez-vous sur le site officiel de Python : http://python.org/. Allez dans la rubrique 'Download'.
- Sélectionnez l'une des dernières versions Python 3.x. Téléchargez la en format .tar.bz2.
- Tapez les commandes suivantes :

```
tar -jxvf Python-VERSION.tar.bz2
cd Python-VERSION
./configure
make altinstall
```

La dernière commande permet d'éviter que la version que vous installez n'entre en conflit avec une autre version déjà installée sur votre système.

Lancer l'interpréteur

- Sous Windows :

  Allez dans Menu Démarrer -> Python 3.2 -> Python Command Line

  Ou tapez la commande 'python' dans 'Exécuter'.

- Sous Linux :

  Utilisez la commande 'python'.

  Vous obtenez une ligne de commande semblable à celle ci mais adaptée

  à votre système d'exploitation :

```
Python 2.4.3 (#1, Sep 3 2009, 15:37:12)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-46)] on linux2
Type "help", "copyright", "credits" or "license" for more
  information.
>>>
```

Utilisation de l'interpréteur

L'interpréteur de commande Python permet d'entrer des lignes d'instructions et de les tester au fur et à mesure de leur écriture.
Par exemple si l'on utilise la fonction print() qui permet d'afficher un message à l'écran, il est possible d'entrer :

```
print("Hello world !")
```

Le message 'Hello world !' s'affiche bien sur la console.
Cependant utiliser l'interpréteur de cette façon ne permet pas d'enregistrer les lignes de code produites afin de réaliser un programme.

Quitter l'interpréteur

Pour quitter l'interpréteur il suffit de :

- Sous Windows, taper 'exit()', quit(), Ctrl+Z
- Sous Linux, taper 'exit()' ou faîtes Ctrl+D

Syntaxe

Blocs de code

La plupart des structures en Python s'organisent sous la forme de blocs d'instruction.
La différence entre Python et C, par exemple, est que les blocs ne sont pas délimités par un caractère spécial (par exemple des accolades) mais seulement en fonction de l'indentation.
L'indentation est donc très stricte en Python. Dans un même bloc, toutes les instructions doivent être indentées de la même façon.

Exemple :

```
if True:
   print("True")
else:
   print("False")
```

Cet exemple est correct, il comporte trois blocs : le premier correspondant au 'if' et au 'else', le second correspondant aux instructions à l'intérieur du 'if' et le troisième aux instructions à l'intérieur du 'else'.

Syntaxe

Blocs de code

```
if True:
    print("True")
    print("Good !")
else:
    print("False")
        print("Try again !")
```

Le troisième bloc de cet exemple engendrera une erreur car l'indentation est différente entre les deux instructions.

Syntaxe

Commentaires

Pour écrire un commentaire en Python, il faut introduire le caractère # au début de la ligne.
Ainsi toute la ligne est en commentaire.
Un commentaire peut aussi être introduit après une instruction, il doit toujours être précédé d'un #.

Exemples :

```
print("Hello Pythagore. ")
#Ceci est un commentaire.
print("Comment allez-vous?") #Ceci est aussi un commentaire.
```

Syntaxe

Quelques règles

- Pour mettre plusieurs instruction sur la même ligne il faut les séparer par un point-virgule.

```
print("Hello Pythagore. "); print("Comment allez-vous ?")
```

- Il est possible d'écrire une instruction sur plusieurs lignes si chaque ligne est terminée par un \.

```
total = objet_un + \
        objet_deux + \
        objet_trois
```

- Un bloc doit forcément contenir au minimum une instruction, sinon il génèrera une erreur.

Un programme en Python

Un programme est un ensemble d'instructions.

Pour écrire vos programmes, utilisez un éditeur de texte tel que vim (sous Linux), IDLE, notepad++...

L'extension des programmes exécutables par l'interpréteur Python est .py

En-tête du programme

```
#!/usr/local/bin/python
```

Cette ligne indique l'emplacement de Python sur votre système. Elle peut être différente selon l'endroit où vous avez effectué l'installation.

Il est aussi possible de rajouter une ligne pour définir l'encodage utilisé. Cette ligne est facultative, l'encodage par défaut est UTF-8. Si l'encodage nécessaire était, par exemple, latin-1 alors il faudrait écrire :

```
# -*- coding: latin-1 -*-
```

Exécution d’un programme

Pour exécuter le programme ‘test.py’, tapez la ligne de commande :

```
python test.py
```

II. Notions de base

alaa.elyassir@gmail.com

Principaux types de données

Les nombres

- Les entiers : type 'int'
- Les nombres à virgule flottantes : type 'float‘
- Les nombres complexes : type ‘complex’

Les booléens

Les booléens sont des variables qui peuvent prendre la valeur 'True' (pour 'vrai') et 'False' (pour 'faux').

Les chaînes de caractères

De type 'string'.

En Python, les chaînes de caractères s'écrivent de quatre façons différentes :

- Entre guillemets ("ceci est une chaîne de caractères")
- Entre apostrophes ('ceci est une chaîne de caractères')
- Entre triples guillemets ("""ceci est une chaîne de caractères""")
- Entre triples apostrophes ('''ceci est une chaîne de caractères'''‘)

Principaux types de données

Les séquences

- Les listes : suite modifiable et ordonnée de valeurs quelconques
  - Exemple : [1, 2, 3]
- Les tuples : suite non modifiable et ordonnée de valeurs quelconques
  - Exemple : (1, 2, 3)
- Les range : un itérateur qui retourne une suite ordonnée de valeur
  - Exemple : range(5) # retourne tous les entiers de 0 à 4

Les maping type

- Les dictionnaires : Collection de couples (clé, valeur)
  - Exemple : {1: 4, 4: 8, 9: 10}

Les ensembles

- Les set : Ensemble de valeurs uniques
  - Exemple : {1, 2, 3}

Les variables

Noms de variables

- Le nom de la variable ne peut être composé que de lettres (non accentuées), de chiffres et du symbole souligné "_".
- Le nom de la variable ne peut pas commencer par un chiffre.
- Le langage Python est sensible à la casse, ce qui signifie que des lettres majuscules et minuscules ne constituent pas la même variable (AGE est différent de age).
- Par convention, les noms de variable doivent être explicites.
- Souvent, la variable commence par une minuscule, chaque nouveau mot commence par une majuscule.
- *Exemples : maVariable, essai_2,...*

Affecter une variable

Les variables n'ont pas besoin d'être déclarées, il est possible de leur donner directement une valeur.
C'est ce qu'on appelle le typage dynamique. En effet, il n'est pas nécessaire de donner le type de la variable au préalable. Lors de l'affectation, la variable prend automatiquement le type correspondant.

Exemple : Pour donner une valeur (par exemple 10) à une variable, il suffit d'écrire :

```
nomVariable = 10
```

Exemples :

```
a = 2 # a est de type int
b = 3.50 # b est de type float
chaine = "Bonjour" # chaine est de type string
```

Affectation multiple

Il est possible de déclarer plusieurs variables en même temps sur la même ligne :

```
a, b, chaine = 3, 23, 'coucou'
# a vaut 3, b vaut 23 et chaine vaut 'coucou'
x = y = 10
# x et y valent 10
```

Mots clés

Les mots clés sont des mots qui sont réservés, vous ne pouvez donc pas appeler une variable avec l'un de ces mots.

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

La plupart de ces mots clés seront explicités par la suite.

alaa.elyassir@gmail.com

Opérateurs

Les relations entre différents termes sont exprimées à l'aide d'opérateurs (par exemple : +, <, =, …).

Priorité

Lorsqu'il y a plus d'un opérateur dans une expression, l'ordre dans lequel les opérations doivent être effectuées dépend de règles de priorité. Avec Python, les règles de priorité sont les mêmes que les règles mathématiques. Si deux opérateurs ont la même priorité, l'évaluation est effectuée de gauche à droite.

Associativité

Lorsque deux opérateurs ont la même précédence c'est l'associativité qui détermine l'ordre d'exécution des opérations.

Les opérateurs sont en général associés de gauche à droite, c'est-à-dire que les opérateurs avec la même priorité sont évalués de la gauche vers la droite. Cependant certains opérateurs comme les opérateurs d'assignation ont une associativité de droite à gauche.

Exemples : 2+3+4 est évalué comme (2+3)+4

a=b=c est traité comme a=(b=c)

Il est conseillé de placer des parenthèses lorsqu'un doute peut exister sur l'ordre des opérations.

Divers opérateurs

Opérateurs d'affectation

En plus des opérateurs classiques, il existe des opérateurs d'affectation combinés : +=, -=, *=, /=.

Exemples :

```
a, b, c = 2, 5, 10
a += 3 # équivaut à a = a + 3, donc a vaut maintenant 5
b *= 4 # b vaut maintenant 20
c /= 2 # c vaut maintenant 5
```

Opérateurs logiques

Les opérateurs logiques renvoient une valeur booléenne 'True' ou 'False' en fonction de l'évaluation de l'expression :

- a and b : ET logique booléen, si a est faux, retourne a et b non évalué, sinon retourne b
- a or b : OU logique booléen, si a est vrai, retourne a et b non évalué, sinon retourne b
- not a : NON logique, si a est faux, retourne 'True', sinon retourne 'False‘

Divers opérateurs

Opérateurs de comparaison

Tout comme les opérateurs logiques, les opérateurs de comparaison renvoient une valeur booléenne 'True' ou 'False'.

Les opérateurs de comparaison sont : **<, >, <=, >=, == pour égal, != pour différent**

Exemple :

a = 1<3 # a vaut 'True' car 1 est inférieur à 3.

Opérateurs

Quelques opérateurs classiques

**	puissance
* / + -	opérateurs mathématiques standard
//	division entière (5//3 donne 1)
%	modulo, reste (10%3 donne 1)
< > <= >= == !=	comparaison
= += -= *= %= /=	opérateurs d'affectation
not and or	opérateurs logiques

Quelques fonctions utiles

Fonction print()

La fonction print() permet d'afficher des données dans la console. Cela peut être une chaîne de caractère ou une variable.

Exemple :

```
a = 15
print("Bonjour !") # Affiche Bonjour !
print(a) # Affiche la valeur de a, donc 15
```

Fonction input()

La fonction input() permet de récupérer une valeur saisie par l'utilisateur.

Cette fonction peut prendre en paramètre une chaîne de caractères indiquant à l'utilisateur ce qu'il doit écrire.

Exemple :

```
nombre = input("Entrez un nombre :")
print("Vous avez écrit le nombre suivant :", nombre)
```

Structures conditionnelles

Structure de test : if

La structure de test s'écrit avec if :

```
if expression_à_tester :
   bloc1
else : #si l'expression_à_tester est fausse
   bloc2
```

Le bloc else est facultatif, il est possible de n'avoir que le premier bloc, dans ce cas il ne se passe rien si l'expression à tester est fausse.

Exemple : le but est de diviser 100 par une variable a

```
if a != 0 :
   print(100/a)
else :
   print("la division par zéro est interdite!")
```

Structure de test : if et elif

On peut ajouter une autre condition au milieu :

```
if expression_à_tester :
   bloc1
elif expression_à_tester_2 :
   bloc2
else : #si les deux expressions à tester sont fausses
   bloc3
```

Il est possible d'ajouter plusieurs blocs elif les uns à la suite des autres s'il y a plus de deux expressions à tester.

Exemple : déterminer si un nombre est inférieur, égal ou supérieur à 50

```
if unNombre < 50 :
   print("Votre nombre est plus petit que 50.")
elif unNombre == 50 :
   print("Votre nombre est exactement 50")
else :
   print("Votre nombre est plus grand que 50.")
```

Boucle : while

Il est possible d'effectuer une boucle avec 'while' :

```
while condition :
   instructions...
```

Tant que la condition est vraie, le programme parcourt le bloc

Exemple : écrire les chiffres de 1 à 9

```
unNombre = 1
while unNombre < 10 :
    print("J'apprends à compter :")
    print(unNombre)
    unNombre += 1
```

Boucles : mots clés

Mot clé 'break'

Le mot clé 'break' s'ajoute dans une boucle afin de la casser.

```
while condition :
   if condition2 :
     instruction
     break # Le break interrompt la boucle while.
   else :
     instruction2
```

Exemple : une personne entre un certain nombre de données, une entrée vide signifiant que la liste de données est finie.

```
print("Entrez des données. (Entrée vide pour arrêter).")
while True :
   entree = input()
   if entree == "" :
     print("Nous avons terminé !")
     break
   else :
     print("Vous avez entré :",entree)
```

Boucles : mots clés

Mot clé 'continue'

Le mot clé 'continue' permet d'interrompre la boucle et de passer à l'itération suivante.

```
while condition :
   # 'continue' renvoie ici
    if condition2 :
       instruction
       continue
    instruction2
```

Exemple : écrire tous les nombres pairs inférieurs à 50

```
    unNombre = 0
    while unNombre < 50 :
        unNombre += 1
        if unNombre % 2 != 0: # signifie que le nombre n'est pas divisible
                              # par 2
          continue # si le nombre est impair alors le programme passe à
                   # l'itération suivante
        print("unNombre =",unNombre) # sinon le programme écrit le nombre
```

III. Les séquences

str, list, tuple et range

alaa.elyassir@gmail.com

Les séquences

Suite ordonnées de valeur qui disposent de certaines opérations

Operation	Result	Notes
`x in s`	`True` if an item of *s* is equal to *x*, else `False`	(1)
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`	(1)
`s + t`	the concatenation of *s* and *t*	(6)(7)
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times	(2)(7)
`s[i]`	*i*th item of *s*, origin 0	(3)
`s[i:j]`	slice of *s* from *i* to *j*	(3)(4)
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*	(3)(5)
`len(s)`	length of *s*	
`min(s)`	smallest item of *s*	
`max(s)`	largest item of *s*	
`s.index(x[, i[, j]])`	index of the first occurrence of *x* in *s* (at or after index *i* and before index *j*)	(8)
`s.count(x)`	total number of occurrences of *x* in *s*	

https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range

alaa.elyassir@gmail.com

Chaines de caractères

Rappels

Les chaînes de caractères sont de type 'string'.

En Python, elles s'écrivent de quatre façons différentes :

- Entre guillemets ("ceci est une chaîne de caractères")
- Entre apostrophes ('ceci est une chaîne de caractères')
- Entre triples guillemets ("""ceci est une chaîne de caractères""")
- Entre triples apostrophes ('''ceci est une chaîne de caractères''')

Le symbole \ permet :

- d'insérer des caractères spéciaux : par exemple l'apostrophe s'il est utilisé comme délimiteur.
- d'insérer un caractère interprété : par exemple \n correspond à une nouvelle ligne ou \t correspond à une tabulation.
- d'écrire sur plusieurs lignes.

Exemples :

```
print('Je m\'appelle Pythagore \n')
print(" \"Ceci est une citation \
très longue \", AUTEUR")
print("une tabulation\tlaisse un vide")
```

Découpage de chaînes de caractères

Une chaîne de caractères est composée d'un certain nombre de caractères auxquels il est possible d'accéder séparément.

Accès à un caractère

Pour accéder à un caractère bien déterminé, il faut utiliser sa position dans la chaîne placée entre crochets. Remarquez cependant que les caractères sont numérotés à partir de zéro.

```
nom_chaine[position]
```

Cette syntaxe donne accès à un caractère mais ne permet pas de le modifier.

Exemple :

```
chaine = "voici un exemple"
print(chaine[0], chaine[3]) # renvoie 'v, c‘
print(chaine[-1], chaine[-3]) # renvoie ‘e, p'
```

Découpage d'une chaîne

Il est possible d'extraire une partie d'une chaîne de caractères :

nom_chaine[début:fin]

Donne les caractères d'indice i tel que 'debut <= i < fin', le caractère d'indice égal à fin n'est donc pas pris en compte

Exemples :

```
chaine = "voici un exemple"
print(chaine[:3]) # renvoie 'voi', équivalent à chaine[0:3]
print(chaine[2:5]) # renvoie 'ici‘
print(chaine[2:5:2]) # renvoie 'ii'
print(chaine[7:]) # renvoie 'n exemple‘
print(chaine[:7]) # renvoie ‘voici u‘
print(chaine[2:5:-1]) # renvoie ''
print(chaine[-4:]) # renvoie ‘mple'
```

Manipulation de chaînes

Concaténation

L'opérateur de concaténation pour les chaînes de caractères est le +.

chaine1 + chaine2

Exemple :

```
chaine1 = "Bonjour"; chaine2 = "Pythagore"
chaine3 = chaine1 + " " + chaine2
print(chaine3) # Renvoie 'Bonjour Pythagore'
```

Test d'appartenance

Il est possible de tester l'appartenance d'un ensemble de caractères à une chaîne en utilisant in ou not in.

Ces opérateurs permettent de renvoyer un booléen.

Exemples :

```
chaine = "voici un exemple"
print("emp" in chaine) # Renvoie True, "emp" est dans la chaîne
print("aze" in chaine) # Renvoie False, 'aze' n'est pas dans la chaîne
print("aze" not in chaine) # Renvoie True, 'aze' n'est pas dans la
   chaîne
```

Manipulation de chaînes de caractères

Conversion

Si une chaîne représente un nombre, il est possible de la convertir en nombre avec la fonction int()

```
chaine = '201'
n = int(chaine)
print(n+20) # Renvoie 221
```

Remplacement d'une séquence dans une chaîne

La méthode replace permet de remplacer un texte par un autre dans une chaîne de caractères :

```
chaine.replace(texte1, texte2[, nbmax])
```

texte1 : texte à remplacer

texte2 : texte de remplacement

nbmax : argument optionnel permettant de préciser le nombre de remplacements à faire

Exemple :

```
chaine = "Fischers Fritz fischt frische Fische"
print chaine.replace("sch", "tz") # renvoie Fitzers Fritz fitzt fritze
   Fitze
print chaine.replace("sch", "tz", 2) # renvoie Fitzers Fritz fitzt
   frische Fische
```

Les listes

Définition d'une liste

Une liste se définit avec la syntaxe suivante :

```
nom_liste = [element1, element2, element3]
```

Une liste peut contenir des éléments de types différents, par exemple 'element1' peut être un entier, tandis que 'element2' serait une chaîne de caractères.

Accès à un élément

L'accès à un élément d'une liste utilise la même syntaxe que pour les chaînes de caractères :

```
nom_liste[indice]
```

Il est aussi possible de découper la liste comme pour les chaînes de caractères.

L'indice -1 correspond au dernier élément de la liste.

Chaque élément peut alors être modifié dès que son indice est connu.

Exemples :

```
liste1 = ['lundi', 'mardi', 'mercredi']
liste2 = [0, 1, 2, 3, 54, 102]
liste3 = ['Robert',26,'anglais']
print(liste1[0]) # Renvoie 'lundi'
print(liste2[-1]) # Renvoie 102
liste3[1] += 2
print(liste3) # Renvoie ['Robert',28,'anglais']
print(liste1[:2]) # Renvoie ['lundi', 'mardi']
```

Les références

Lorsque vous affectez une valeur à une variable en Python, tout est stocké sous forme de références et non sous forme de copies.

Exemple :

```
L = [1,2,3]
M = ['x',L,'y']
print(M) # Renvoie ['x', [1, 2, 3], 'y']
L[1]=0
print(M) # Renvoie ['x', [1, 0, 3], 'y']
# M contient une référence vers les données de L, donc modifier L modifie aussi M
# Pour créer une véritable copie de la liste, si M ne devait pas être modifiée,
# il faudrait utiliser L[:]
L = [1,2,3]
M = ['x',L[:],'y']
print(M) # Renvoie ['x', [1, 2, 3], 'y']
L[1]=0
print(M) # Renvoie ['x', [1, 2, 3], 'y']
print(L) # Renvoie [1,0,3]
```

Les tuples

Un tuple est une liste dans laquelle il n'est pas possible de modifier les éléments. C'est une séquence non modifiable.

- Pas de suppression d'élément
- Pas d'ajout d'élément
- Pas de modification d'élément

Les tuples sont très similaires aux listes en ce qui concerne la syntaxe à la différence que les crochets deviennent des parenthèses.

Il existe une fonction tuple() qui ressemble à la fonction list().

Exemples :

```
unTuple = tuple("Ok !")
print(unTuple) # Renvoie ('O', 'k', ' ', '!')
tuple2 = ('Ok !', 23, 34, 45, True)
tuple3 = (1,)  # tuple contenant une seule valeur
print(tuple2) # Renvoie ('Ok !', 23, 34, 45, True)>
```

Exemple de tuple utilisé implicitement : l'affectation multiple

```
a,b = 3,4 # Équivaut à (a,b) = (3,4)
```

Range

La fonction range retourne un itérateur qui est également une séquence.
Elle permet de définir un objet par son premier élément, le pas et le dernier élément. Le reste des éléments sont calculés à fur et à mesure.

Exemples:

```
range(10)  # contient tout les entiers de 0 à 9
range(5, 10)  # contient tous les entiers de 5 à 9
range(5, 10, 2)  # contient les entiers 5, 7et 9
```

Manipulation de séquences

Quelques fonctions

Il existe des fonctions intégrées définies pour les séquences.

- len(liste) : renvoie le nombre d'éléments présents dans la liste (donne le nombre de caractères pour une chaîne de caractère).
- min(liste)/max(liste) : retourne le plus petit/grand élément de la séquence (valable aussi pour une chaîne de caractères l'ordre alphabétique est alors considéré)
- sorted(liste) : retourne la même liste mais avec les éléments rangés dans l'ordre croissant.
- reversed(liste) : retourne la liste (un itérateur) inversée
- sum(liste) : retourne la somme de tous les éléments de la liste.

Parcourir une liste

Boucle while

Pour parcourir une liste, il est nécessaire d'utiliser une boucle. Avec une boucle while, la syntaxe serait :

```
i = 0
while i<len(liste) :
    print(liste[i])
    i += 1
```

Boucle for

La boucle for est vivement conseillée pour parcourir la liste. La syntaxe est la suivante :

```
for element in liste :
    print(element)
```

Fonction enumerate

La fonction enumerate() renvoie un itérateur dont les éléments sont des tuples contenant les couples (indice,élément). Elle permet donc d'accéder à l'élément et à son indice en même temps.

```
for i,element in enumerate(liste) :
    liste[i] += 100
    print(liste[i]) # Renvoie l'élément initial + 100
    print(element) # Renvoie l'élément initial
```

for … else

On peut avoir un else à la suite d'un for.

Le bloc else n'est exécuté que si la boucle for se déroule normalement, sans faire appel à break et sans levée d'exception.

Exemple:

```
for e in chaine:
    if e == 'e':
        break
else:
    print('else')

# si chaine contient un 'e', on n'affiche pas la chaine 'else'
# si chaine ne contient pas de 'e', on affiche la chaine 'else'
```

Dict & set

alaa.elyassir@gmail.com

Les dictionnaires

Définition

En Python, un dictionnaire permet de stocker un ensemble de données tout comme le fait une liste ou un tuple. La différence est que les données ne sont pas ordonnées.

Chaque donnée n'est pas associée à un indice mais à une clé. Chaque fois que vous voudrez ajouter une donnée, vous devrez donc l'associer à une clé pour former le couple (clé, valeur). On dit que les dictionnaires sont de la catégorie des 'maps'.

Caractéristiques

- Les dictionnaires sont délimités par des accolades {}
- Les clés doivent être de types non modifiables (entier, chaîne de caractères, tuples) : elles ne doivent donc pas être des variables.
- Un dictionnaire ne peut pas contenir deux clés identiques. En revanche, rien n'empêche d'avoir deux valeurs identiques.

Créer un dictionnaire

Création directe

La syntaxe est la suivante :

```
nomDictionnaire = { cle : valeur, cle2 : valeur2, cle3 : valeur3 }
```

Exemple :

```
dictionnaire = {"Hello !" : "Bonjour !", " Thanks !" : "Merci !", "Bye
   !" : "Au revoir !"}
```

La fonction 'dict'

La fonction 'dict' fonctionne de manière similaire à la fonction 'list' pour les listes. Il y a trois façons de l'utiliser :

- Lui donner en argument des listes à deux éléments qui formeront les couples (clé,valeur) :

```
dictionnaire = dict([["a", 1], ["b", 2], ["c", 3], ["d", 4]])
```

- Lui donner en argument des tuples à deux éléments qui formeront les couples (clé,valeur) :

```
liste = 'abcd'
dictionnaire = dict([(liste[i], i+1) for i in range(4)])
```

- Lui donner en argument des affectations :

```
dictionnaire = dict(a=1, b=2, c=3, d=4)
```

Manipulation des éléments

Ajout ou modification

Pour ajouter ou modifier un couple dans un dictionnaire, il suffit d'indiquer entre crochets la clé à laquelle nous souhaitons accéder. Si la clé n'existe pas, elle est ajoutée au dictionnaire avec la valeur spécifiée. Sinon, l'ancienne valeur est remplacée par la nouvelle.

Exemple :

```
dictionnaire = {}
dictionnaire["pseudo"] = "user"
dictionnaire["mot de passe"] = "*"
```

Accéder à une valeur

Pour accéder à une valeur, il suffit de la trouver à l'aide de sa clé :

```
dictionnaire[cle]
```

Exemple :

```
print(dictionnaire["mot de passe"])
# Renvoie '*'
```

Quelques méthodes

Méthode update()

La méthode update() permet d'ajouter au dictionnaire auquel elle est appliquée, tous les couples (clé,valeur) du dictionnaire passé en argument.

```
dico1 = dict([('abcdef'[i], i) for i in range(6)])
dico2 = dict(clé1=1,clé2=2,clé3=3)
dico1.update(dico2)
print(dico1)
# Renvoie {'a': 0, 'c': 2, 'b': 1, 'e': 4, 'd': 3, 'f': 5, 'clé1':
   1, 'clé2': 2, 'clé3': 3}
```

Quelques méthodes

Effacer des éléments

Il y a trois façons d'effacer des éléments :

- La méthode clear() efface tous les couples du dictionnaire.

```
dico1 = dict([('abcdef'[i], i) for i in range(6)])
dico1.clear()
print(dico1)
```

- La méthode pop(cle) supprime le couple précisé par la clé et renvoie la valeur associée à la clé supprimée :

```
placard = {"chemise":3, "pantalon":6, "tee shirt":7}
placard.pop("chemise") # Renvoie 3
```

- Le mot clé del supprime le couple associée à la clé :

```
placard = {"chemise":3, "pantalon":6, "tee shirt":7}
del placard["chemise"]
```

Quelques méthodes

Méthode values()

La méthode values() renvoie la liste des valeurs contenues dans le dictionnaire :

Exemple :

```
inventaire = {'pommes': 430, 'bananes': 312, 'oranges' : 274, 'poires'
   : 137}
print(inventaire.values())
# Renvoie : dict_values([137, 312, 430, 274])
```

Méthode keys()

La méthode keys() renvoie la liste des clés contenues dans le dictionnaire :

Exemple :

```
inventaire = {'pommes': 430, 'bananes': 312, 'oranges' : 274, 'poires'
   : 137}
print(inventaire.keys())
# Renvoie : dict_keys(['poires', 'bananes', 'pommes', 'oranges'])
```

Parcourir un dictionnaire

Le parcours d'un dictionnaire se fait de façon similaire à celui d'une liste, donc avec une boucle for.

Accéder aux clés

Il est possible d'accéder aux clés de façon très simple :

```
for cle in dictionnaire :
   print("La clé %s est associée à la valeur %d"
   %(cle,dictionnaire[cle]))
```

Accéder aux clé et valeurs

La méthode items permet d'accéder directement aux clé et aux valeurs :

```
for cle, valeur in dico1.items() :
   print("La clé %s est associée à la valeur %d" %(cle,valeur))
```

Test d'appartenance

Tout comme pour les listes, les opérateurs 'in' et 'not in' permettent de vérifier si une clé appartient au dictionnaire.

```
dico1 = dict([('abcdef'[i], i) for i in range(6)])
'a' in dico1 # Renvoie True
't' in dico1 # Renvoie False
't' not in dico1 # Renvoie True
```

Définition de set

- Un objet set est une collection non ordonnée d'objets distincts pouvant être séparés.
- Les utilisations courantes incluent le test d'adhésion, la suppression des doublons d'une séquence et le calcul d'opérations mathématiques telles que l'intersection, l'union, la différence et la différence symétrique. (Pour les autres conteneurs, voir les classes intégrées dict, list et tuple, ainsi que le module collections.)
- Exemple:

```
set1 = {1, 1, 2}  # set1 correspond à {1, 2}
set2 = set([2, 2, 3])  # set2 correspond à {2, 3}
```

Méthodes de set

- `len(s)` # Donne le nombre d'éléments dans le *set s* (cardinalité de *s*).
- `x in s` # Test d'appartenance de *x* dans *s*.
- `x not in s` # Test de non-appartenance de *x* dans *s*.
- `isdisjoint(other)` # Renvoie True si l'ensemble n'a aucun élément en commun avec *other*. Les ensembles sont disjoints si et seulement si leurs intersection est un ensemble vide.
- `issubset(other)` # (set <= otherTeste) si tous les éléments du set sont dans *other*.
- `set < other` # Teste si l'ensemble est un sous-ensemble de *other*, c'est-à-dire, set <= other and set != other.
- `issuperset(other)` # set >= other Teste si tous les éléments de *other* sont dans l'ensemble.
- `set > other` # Teste si l'ensemble est un sur-ensemble de *other*, c'est-à-dire, set >= other and set != other.
- `union(*others)` # set | other | ...Renvoie un nouvel ensemble dont les éléments viennent de l'ensemble et de tous les autres.
- `intersection(*others)` # set & other & ...Renvoie un nouvel ensemble dont les éléments sont commun à l'ensemble et à tous les autres.
- `difference(*others)` # set - other - ...Renvoie un nouvel ensemble dont les éléments sont dans l'ensemble mais ne sont dans aucun des autres.
- `symmetric_difference(other)` # set ^ other Renvoie un nouvel ensemble dont les éléments sont soit dans l'ensemble, soit dans les autres, mais pas dans les deux.
- `copy()` # Renvoie un nouvel ensemble, copie de surface de *s*.

Méthodes de set

- update(**others*) # set |= other | ...Met à jour l'ensemble, ajoutant les éléments de tous les autres.
- intersection_update(**others*) # set &= other & ...Met à jour l'ensemble, ne gardant que les éléments trouvés dans tous les autres.
- difference_update(**others*) # set -= other | ...Met à jour l'ensemble, retirant les éléments trouvés dans les autres.
- symmetric_difference_update(*other*) # set ^= otherMet à jour le set, ne gardant que les éléments trouvés dans un des ensembles mais pas dans les deux.
- add(*elem*) # Ajoute l'élément *elem* au set.
- remove(*elem*) # Retire l'élément *elem* de l'ensemble. Lève une exception KeyError si *elem* n'est pas dans l'ensemble.
- discard(*elem*) # Retire l'élément *elem* de l'ensemble s'il y est.
- pop() # Retire et renvoie un élément arbitraire de l'ensemble. Lève une exception KeyError si l'ensemble est vide.
- clear() # Supprime tous les éléments du *set*.

https://docs.python.org/fr/3.7/library/stdtypes.html#set

Expression de compréhension

alaa.elyassir@gmail.com

Liste de compréhension

Les listes de compréhension permettent d'effectuer les deux opérations

- filtrer : en récupérant uniquement les éléments vérifiant une certaine condition
- mapper : en appliquant une fonction à l'ensemble des éléments (filtrés)

Syntaxe:

```
nouvelle_liste = [function(item) for item in liste if condition(item)]
```

Exemple:

```
entiers = [1, 3, 5, 6, 9]
multiples_trois_au_carre = [e**2 for e in entiers if e%3==0]
```

variante:

```
nouvelle_liste = [f(e) if cdt(e) else g(e) for e in liste]
```

Dict & set compréhension

- Dict
  - `{x: f(x) for x in seq if cdt(x)}`
- Set
  - `{f(x) for x in seq if cdt(x)}`

IV. Les Fonctions

alaa.elyassir@gmail.com

Présentation

Définition

La fonction est composée de

- Son nom
- Ses paramètres ou arguments : données/variables qui sont passés à la fonction
- Sa sortie : la ou les valeurs retournées par la fonction
- ➢ En principe, une fonction retourne une seule valeur, mais avec Python, il est possible de retourner un 'tuple', qui est un ensemble de valeurs.

Avantages

Il y a deux principaux avantages à l'écriture de fonctions :

- Le code est alors réutilisable. S'il est nécessaire de faire la même démarche plusieurs fois, la fonction permet de définir la démarche puis de l'utiliser par un simple appel.
- Décomposition du code. Les fonctions permettent de décomposer le code en morceaux qui ont des rôles bien définis.

Déclarer et appeler une fonction

Déclarer

Pour déclarer une fonction, il faut utiliser la syntaxe suivante :

```
def nom_fonction(parametres) :
    instructions
    return sortie
```

'parametres' correspond à ce que l'on donne en entrée à la fonction. L'opérateur 'return' permet de renvoyer une valeur 'sortie'.

Appeler une fonction

La définition de la fonction doit toujours précéder son utilisation.

Pour utiliser une fonction, il suffit de l'appeler en utilisant son nom et en lui fournissant les paramètres dont elle a besoin.

```
variable = nom_fonction(parametres)
```

- Remarque : une fonction peut ne pas avoir de paramètres ou ne pas retourner de valeur selon le but recherché.

Portée des variables

Les variables définies dans une fonction ont une portée limitée à la fonction. A contrario, une variable définie dans le programme général peut être utilisée dans une fonction.

Exemple :

```
a = 10
def fonction() :
    a = 5
    print(a)

def fonction2() :
   print(a)

fonction(); fonction2() # Renvoie 5, 10
```

Lorsqu'une variable est utilisée, elle est d'abord cherchée localement puis ensuite si elle n'est pas trouvée, elle est cherchée globalement. C'est pour cela que fonction() renvoie 5 et non 10.

Mot clé global

Le mot clé global permet de préciser qu'une variable appartient à l'espace mémoire général et non à celui de la fonction. Cependant l'utilisation de ce mot clé n'est pas toujours recommandée.

Exemple :

```
def fonction() :
   global a
   a = 5
   print(a)

def fonction2() :
   print(a)

fonction(); fonction2() # Renvoie 5, 5
```

Passage d'arguments

Les arguments sont passés aux fonctions par affectation (référence).

- Les arguments non modifiables ne sont pas affectés par la fonction.

  Exemple :

```
def  f(x):
   x += 1
a = 2; f(a); print(a) # Renvoie 2
```

- Les arguments modifiables peuvent être affectés par la fonction.

  Exemple :

```
def g(x):
   x[0] += 1
a = [5]; g(a); print(a) # Renvoie [6]
```

Une liste étant un objet modifiable, lorsque la liste correspondante à 'x' est modifiée, alors a l'est aussi.

Pour modifier un nombre il faut alors, soit utiliser une liste à un élément, soit utiliser la syntaxe suivante :

```
a = 2
def f(x):
   x += 1
   return x
a = f(a); print(a)
```

Les arguments

Valeur par défaut

Il est possible de donner une valeur par défaut à certains ou à tous les paramètres. Ainsi, si la fonction est appelée sans donner de valeur à ce paramètre, c'est la valeur par défaut qui sera utilisée. **La seule** contrainte est que les arguments auxquels une valeur par défaut sera associée doivent nécessairement être déclarés après ceux qui n'en possèdent pas (appelés arguments obligatoires).

Exemple : fonction permettant de mettre un nombre à une certaine puissance

```
def puissance(nombre, exposant = 2) :
   return  nombre ** exposant

print(puissance(3,3)) # Renvoie 27
print(puissance(3)) # renvoie 9
```

Les arguments

Appel désordonné

Les arguments de la fonction peuvent ne pas être mis dans l'ordre lors de l'appel. Cependant il faut alors utiliser l'étiquette de l'argument : l'étiquette est le nom qui est associé à l'argument lors de la déclaration de la fonction.

Exemple : appel de la fonction puissance définie précédemment

```
print(puissance(exposant = 4, nombre = 3)) # Renvoie 81
```

Les arguments

Nombre indéfini d'arguments

Il est possible, dans certains cas, de définir une fonction avec un nombre d'arguments arbitraire.

Par exemple, nous souhaitons écrire un certain nombre de données mais le nombre peut varier d'un appel à l'autre de la fonction. Les arguments seront alors passés sous forme d'un tuple, pour préciser que

le nombre est arbitraire il faut ajouter le symbole '*' devant l'étiquette du tuple.

Exemples :

```
def fonction(*arguments) :
   for element in arguments :
     print(element)
fonction(43, 38, "Bonjour !", True) # Renvoie 43 38 Bonjour ! True

def fonction2(a,b,*arguments) :
   somme = a+b
   print(somme)
   for element in arguments :
     print(element)
fonction2(4, 8, "Bonjour !", True, 102, 1.5)
```

arguments **kwargs

alaa.elyassir@gmail.com

Récursivité

Les fonctions en Python peuvent être récursives, c'est-à-dire faire appel à elle-même.

Les fonctions récursives ont besoin d'un 'cas d'arrêt', plus explicitement cela correspond au cas où la fonction ne va plus s'appeler elle-même. Sans ce cas d'arrêt la fonction entre dans une boucle infinie.

Exemple : Calculer la factorielle d'un nombre : n!.

Rappel : n! = n(n-1)!*

```
def factoriel(nombre) :
     if  nombre == 0 : # C'est le cas d'arrêt
              return 1
     else :
              return nombre * factoriel(nombre - 1)
print(factoriel(2))
```

*Lorsque la fonction factoriel(2) est appelée, alors comme 2 est différent de 1 le programme calcule 2*factoriel(1).*

*L'appel à factoriel(1) donne 1 grâce à la condition d'arrêt. Donc 2*factoriel(1) donne alors 2.*

Fonctions anonymes

Définition

- Les fonctions Lambda sont des fonctions anonymes déclarées au milieu du code. Elles ne sont pas déclarées avec la même syntaxe, en effet il faut utiliser le mot clé 'lambda'.
- Comme la fonction est anonyme, pour pouvoir l'appeler il est nécessaire de la stocker dans une variable ou de l'utiliser directement dans une expression.
- Les fonctions Lambda sont généralement très simples, elles ne nécessitent pas d'utiliser 'return', elle renvoie automatiquement le résultat.

Les fonctions Lambda

Utilisation

Ces fonctions sont particulièrement utiles lors de l'utilisation d'une fonction s'appliquant à une fonction.

- La fonction filter(fonction,séquence) permet de filtrer une séquence en ne gardant que les éléments pour lesquels la fonction renvoie True.

Exemple : Trier les éléments qui contiennent la lettre c

```
liste = ['abc', 'coucou', 'bonjour', 'au revoir', 'ciao']
liste = filter(lambda chaine : 'c' in chaine, liste)
for elem in liste :
   print(elem) # Renvoie 'abc','coucou','ciao'
```

- La fonction map(fonction,liste) permet d'appliquer une opération à chaque élément de la liste.

Exemple : Ajouter 100 à tous les éléments d'une liste.

```
liste = [11, 45, 25, 69]
liste = map(lambda x : x+100, liste)
for elem in liste :
   print(elem) # Renvoie 111, 145, 125, 169
```

Fonctions intégrées

Il existe de nombreuses fonctions intégrées en Python.

Nous en avons déjà vu ou utilisé beaucoup, par exemple :

- print()
- input()
- min()
- max()
- len()
- sorted()
- …

Il en existe encore bien d'autres, et nous en verrons une grande partie au fur et à mesure.

Dès que vous ne savez pas comment utiliser une fonction intégrée, vous pouvez consulter cette page du site officiel de Python : http://docs.python.org/py3k/library/functions.html

La fonction print()

Présentation

La fonction print() permet d'écrire des données :

print ([object,...], sep=' ', end='\n')

- [objet,...] est une liste contenant ce que vous voulez écrire
- sep est le séparateur entre chacun de vos objets, par défaut c'est un espace
- end est ce que la fonction fait après avoir fini la liste, par défaut c'est un retour à la ligne

Exemple :

```
liste = ('Pommes', 'Oranges', 'Bananes')
print(*liste, sep=", ", end=".\n")  # Ce qui donne : 'Pommes, Oranges, Bananes.'
```

Utiliser des variables

Les variables peuvent être écrites tout simplement avec la syntaxe :

```
variable = "ok"
print(variable)
```

Mais il est aussi possible de les introduire au milieu d'une phrase avec le symbole %

```
print("c'est bon, c'est %s" % variable)
print("ce sera fini dans %d %s" % (3, "minutes"))
```

%s correspond à une chaîne de caractères, %d correspond à un entier et %f correspond à un réel.

Instruction yield

L'utilisation de l'instruction yield dans une fonction permet de retourner un générateur. Elle agit ensuite, pratiquement comme un return, à chaque iteration de yield, la fonction s'arrête et renvoie une valeur.
A chaque appel de next(), la fonction reprend là où elle s'était arrêtée. Lorsque le générateur est arrivé au dernier élément, celui ci lève une exception de type StopIteration. L'instruction yield est souvent utilisée pour créer un iterateur sur une liste. *Exemple : un iterateur sur une liste de nombres*

```
>>> def iterateur(liste):
...     while True:
...         for i in liste:
...             yield i
...
>>> liste = (1,2,3)
>>> it = iterateur(liste)
>>> print(next(it))
1
>>> print(next(it))
2
>>> print(next(it))
3
```

V. Programmation objet

alaa.elyassir@gmail.com

Rappels : la programmation objet

Identité

- Les données sont quantifiées en entités discrètes : les objets.
- Les objets peuvent être concrets ou conceptuels.
- Chaque objet possède sa propre identité : deux objets sont distincts même si leurs attributs ont des valeurs identiques.

Exemples :

un fichier, une politique d'ordonnancement, une voiture, une icône

Classification

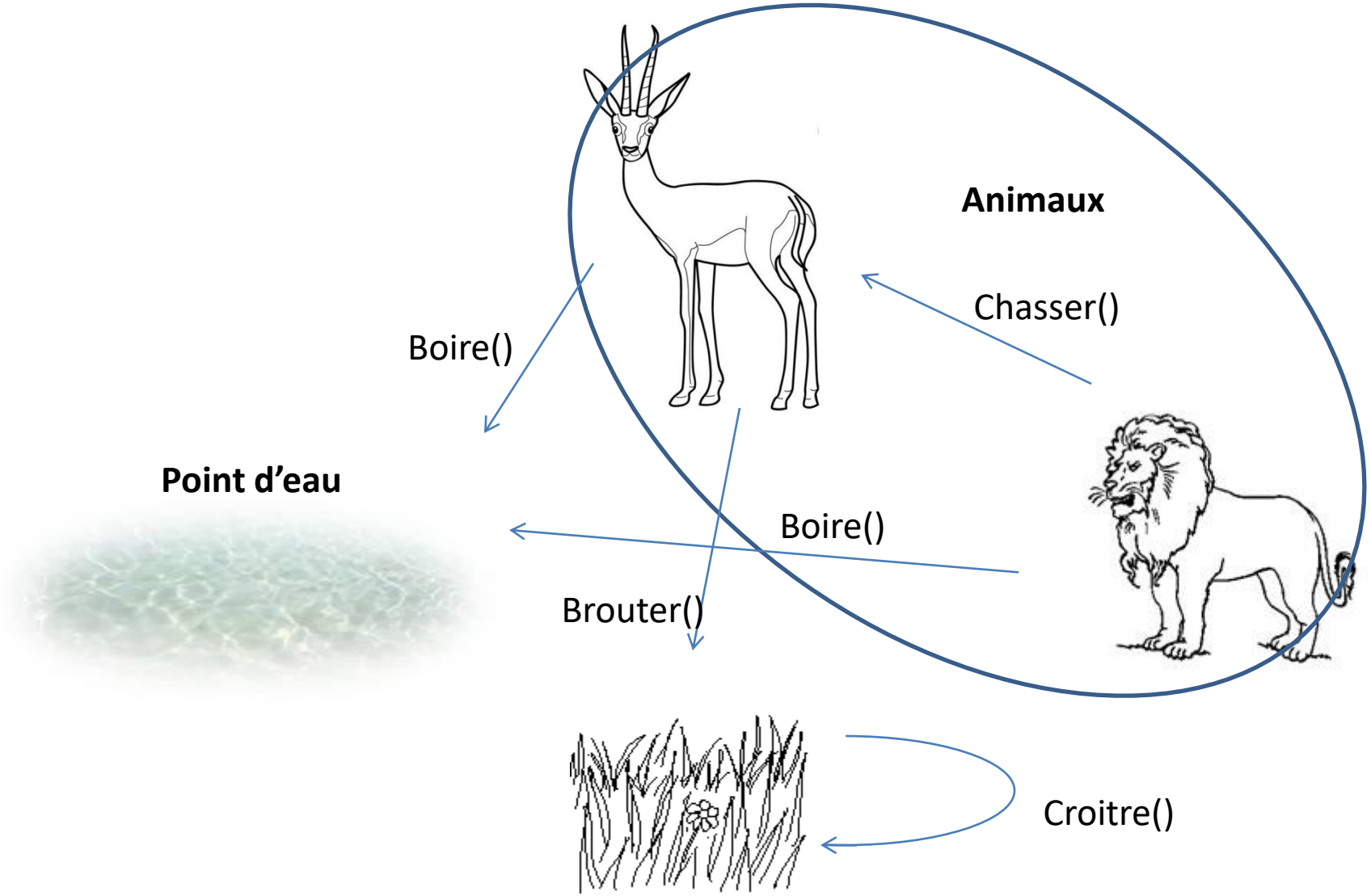
- Les objets ayant la même structure de données (attributs) et le même comportement (opérations) sont regroupés en une classe.
- Chaque classe décrit un ensemble d'objets individuels.
- Chaque objet est dit une instance de sa classe.
- Chaque instance de la classe possède ses propres valeurs pour chaque attribut mais partage les noms d'attributs et les opérations avec les autres instances de la classe.

Exemples :

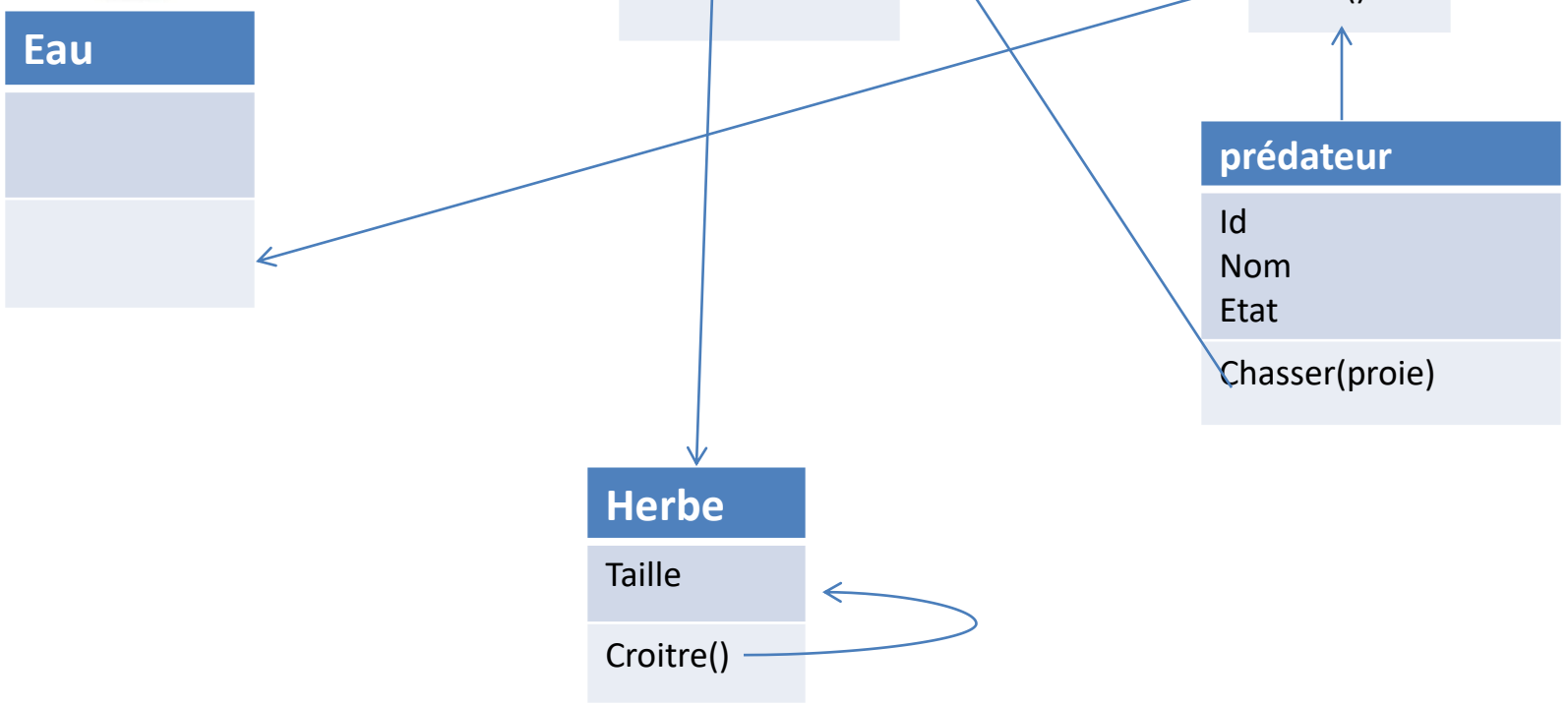
Classe "courbe": regroupe les histogrammes, les courbes y=f(x)

Classe "moyens de transport": regroupe l'avion, le train, la voiture, le vélo

Cas de figure

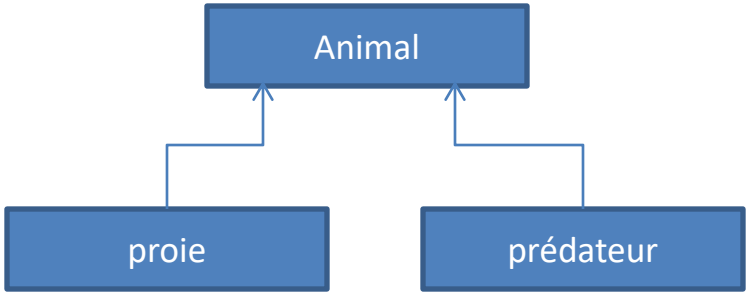


Cas de figure

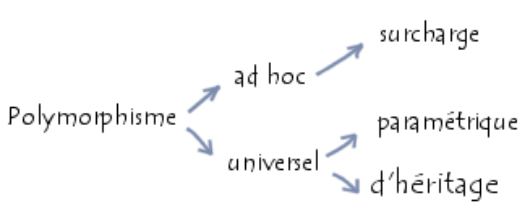


Fondamentaux POO

L'héritage entre classes, permet d'inclure les caractéristiques d'une classe supérieure
Exemple : Les classes Proie et prédateur héritent de la classe Animal, en l'occurrence la méthode boire()



Polymorphisme consiste à fournir une interface unique à des entités pouvant avoir différents types(classes)



Source:http://www.commentcamarche.net/contents/811-poo-le-polymorphisme

Exemple : Addition (+) change selon la classe.

```
1+2
'ceci est' +' un texte'
```

L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés. L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet
Elle permet également d'effectuer d'autres tâches que de déterminer une valeur ou la modifier

Les classes en Python

• En Python, tout est objet : les chaînes de caractères, les listes, …

• Par convention, les noms de classes commencent souvent par une majuscule et ne comportent pas le caractère souligné '_'.

• L'encapsulation des données consiste à garder privée certaines données de l'objet. Il n'est pas possible d'accéder directement à ces données. Il est donc nécessaire de créer des accesseurs qui vont permettre d'obtenir la valeur d'un attribut et des mutateurs qui vont permettre de modifier cette valeur.

Définition des éléments d'une classe

- Une classe est créée par l'instruction 'class'
- Pour créer une instance d'une classe il faut utiliser un constructeur
- Les attributs de cette classe sont créés par les affectations
- Les méthodes sont des fonctions

```
class NomClasse :
   def Methode():
     instructions
```

La méthode __init__

« L'initialisateur » sert à créer des instances de la classe. C'est une méthode spéciale qui s'appelle __init__.

La syntaxe d'un constructeur est la suivante :

```
def __init__(self,attribut1,attribut2,...):
   self.attribut1 = attribut1
   self.attribut2 = attribut2
   . . .
```

'self' représente l'instance qui va être créé, ce sera toujours le premier argument du constructeur.

'self.attribut1' permet d'accéder à l'attribut1 de self.

Ainsi le constructeur définit chaque caractéristique de l'instance.

Il suffit alors d'appeler le constructeur pour créer une instance.

Exemple : Définition d'une classe Personne

```
class Personne:
   def __init__(self,nom,prenom,age):
     self.nom = nom
     self.prenom = prenom
     self.age = age
personne1 = Personne('Bachetet','Charles',25)   # L'argument 'self'
   correspond à 'personne1'
```

Encapsulation

Les attributs privés

- Le principe d'encapsulation oblige l'ajout de méthodes telles que les accesseurs et les mutateurs pour les données qui doivent rester privées. Le fait d'avoir la possibilité de rendre un attribut privé permet de faire en sorte qu'un attribut de notre objet ne soit pas modifiable ou par exemple, de pouvoir mettre à jour un autre attribut à chaque modification de l'attribut privé.
- Une donnée privée doit avoir un nom qui commence par un souligné : _age. Il faut savoir que la donnée n'est privée que par convention, il est tout à fait possible d'y accéder directement, mais conventionnellement c'est interdit.
- On définit alors une propriété (property) qui précise que l'attribut est privé. La fonction property attend comme arguments l'accesseur et le mutateur : property(_get_age, _set_age)
- Par convention, les accesseurs doivent avoir un nom qui commence par 'get' et les mutateurs doivent avoir un nom qui commence par 'set'.

Les attributs "privés"

Exemple :

```
class Personne :
    def __init__(self,nom,prenom,age):
        self.nom = nom
        self.prenom = prenom
        self._age = age
    def _get_age(self) :
        return self._age
    def _set_age(self,age) :
        self._age = age
    age = property(_get_age, _set_age)

personne1 = Personne('Bachetet','Charles',25)
print(personne1.nom) # Renvoie 'Bachetet', l'attribut nom n'est pas privé
print(personne1.age)
# Renvoie 25, la propriété renvoie la variable age sur l'accesseur qui
   renvoie lui-même _age
personne1.age=26 # Utilise le mutateur
```

Les méthodes spéciales

Il existe des méthodes spéciales (elles contiennent dans leur nom deux soulignés avant et après) telles que __init__.

En voici quelques autres :

- __repr__ : elle affecte la façon dont est affiché l'objet.
- __str__ : est appelée si vous désirez convertir votre objet en chaîne de caractères.
- les méthodes __add__ sont appelées lorsque le symbole + est utilisé
- __eq__(self, objet_a) : opérateur d'égalité. Retourne True si self et objet_a sont égaux, False sinon.
- …

L'héritage

• Si en plus de personnes, nous voulons créer des étudiants ou des retraités par exemple. Ces classes Etudiant et Retraite vont hériter de la classe Personne, c'est-à-dire qu'elles vont comporter tous les champs et les méthodes de cette classe : un étudiant ou un retraité possède aussi un nom, un prénom et un âge, mais chacun possède des caractéristiques différentes.
• La classe Personne est la classe mère; les classes Etudiant et Retraite sont des classes filles.
• En Python, l'héritage se définit au moment de la création de la classe, la syntaxe pour définir la classe est alors :
class MaClasse(MaClasseMere):
• Une classe fille peut utiliser les méthodes et les attributs de la classe mère et peut aussi avoir ses propres méthodes et attributs.
• Lorsqu'une méthode est appelée sur une instance de la classe fille, le programme va d'abord regarder si elle est définie dans la classe fille, si ce n'est pas le cas, il ira voir les méthodes de la classe mère.
➢Remarque : il est possible d'hériter de plusieurs classes à la fois, c'est l'héritage multiple.

Héritage : exemple

```
class Etudiant(Personne) :
    def __init__(self,nom,prenom,age,identifiant):
        # Utilisation du constructeur de la classe mère
        Personne.__init__(self,nom,prenom,age)
        # Définition d'un nouvel attribut
        self.identifiant = identifiant
    def affiche(self) :
        Personne.affiche(self)
        print("Je suis etudiant, mon identifiant est %d.\n" %
          self.identifiant)
etudiant1 = Etudiant('Bachetet','Aurélie',19,256894)
etudiant1.affiche()
# Affiche :
# "Je m'appelle Aurélie Bachetet et j'ai 19 ans."
# "Je suis etudiant, mon identifiant est 256894."
```

Test de classes

Fonction issubclass()

Cette fonction vérifie si une classe est une sous-classe d'une autre classe. Elle retourne True si c'est le cas, False sinon.

Exemple :

```
print(issubclass(Etudiant, Personne)) # True
print(issubclass(Etudiant, object)) # True
print(issubclass(Personne, object)) # True
print(issubclass(Personne, Etudiant)) # False
```

Fonction isinstance()

Cette fonction permet de savoir si un objet est issu ou hérité d'une classe ou de ses classe-filles.

Exemple :

```
print(isinstance(etudiant1,Etudiant)) # True
print(isinstance(etudiant1,object)) # True
```

Héritage multiple

Il est possible d'hériter de plusieurs classes à la fois, c'est l'héritage multiple.
Dans notre exemple, la classe Fille disposera, dans l'ordre, des méthodes de la classe Mere1 et de la classe Mere2.

```
class Mere1:
    def methode(self):
        print('Mere1')
    ...
```

```
class Mere2:
    def methode(self):
        print('Mere2')
    ...
```

```
class Fille(Mere1, Mere2):
    pass
```

Pour connaitre l'ordre des classes dans lequel une classe fille va chercher les métohdes, on peut utiliser la méthode *mrode la classe type : type.mro(Fille)*