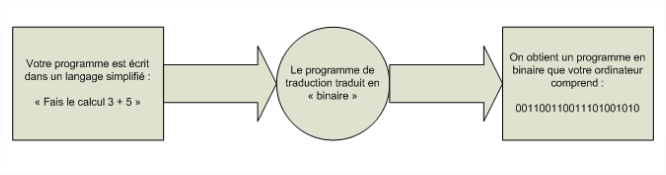
**Apprenez à programmer en python**

Python est un langage de programmation, dont la première version est sortie en 1991. Créé par **Guido van Rossum**, il a voyagé du Macintosh de son créateur, qui travaillait à cette époque au *Centrum voor Wiskunde en Informatica* aux Pays-Bas, jusqu'à se voir associer une organisation à but non lucratif particulièrement dévouée, la **Python Software Foundation**, créée en 2001. Ce langage a été baptisé ainsi en hommage à la troupe de comiques les « Monty Python ».



Python est un langage de programmation **interprété**, c'est-à-dire que les instructions que vous lui envoyez sont « transcrites » en langage machine au fur et à mesure de leur lecture. D'autres langages (comme le C / C++) sont appelés « langages **compilés** » car, avant de pouvoir les exécuter, un logiciel spécialisé se charge de transformer le code du programme en langage machine. On appelle cette étape la « **compilation** ». À chaque modification du code, il faut rappeler une étape de compilation.

Python est un langage de programmation interprété, à ne pas confondre avec un langage compilé.

* Il permet de créer toutes sortes de programmes, comme des jeux, des logiciels, des progiciels, etc.
* Il est possible d'associer des bibliothèques à Python afin d'étendre ses possibilités.
* Il est portable, c'est à dire qu'il peut fonctionner sous différents systèmes d'exploitation (Windows, Linux, Mac OS X,…).

**C'est quoi, une variable ?**

Une variable est une donnée de votre programme, stockée dans votre ordinateur.

Voyez la mémoire de votre ordinateur comme une grosse armoire avec plein de tiroirs. Chaque tiroir peut contenir une donnée ; certaines de ces données seront des variables de votre programme.

**Conventions d'écritures:**

Ecrire la variable en minuscules et à remplacer les espaces éventuels par un espace souligné « \_ ». Si je dois créer une variable contenant mon âge, elle se nommera donc mon\_age. Une autre convention utilisée consiste à passer en majuscule le premier caractère de chaque mot, à l'exception du premier mot constituant la variable. La variable contenant mon âge se nommerait alors monAge.

Certains mots-clés de Python sont **réservés**, c'est-à-dire que vous ne pouvez pas créer des variables portant ce nom.

Ex: And, while, with, pass, lambda, return, etc...

**Les types de données en Python**

Les nombres entiers

Les nombres flottants c'est à dire avec une virgule.

Les chaîne de caractères.

Si vous utilisez les délimiteurs simples (le guillemet ou l'apostrophe) pour encadrer une chaîne de caractères, il se pose le problème des guillemets ou apostrophes que peut contenir ladite chaîne. Par exemple, si vous tapez chaine = 'J'aime le Python!', vous obtenez le message suivant :

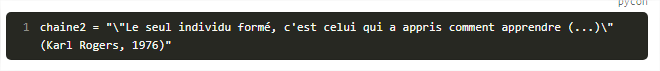


Ceci est du au fait que l'apostrophe de « J'aime » est considérée par Python comme la fin de la chaîne et qu'il ne sait pas quoi faire de tout ce qui se trouve au-delà.  
Pour pallier ce problème, il faut **échapper** les apostrophes se trouvant au cœur de la chaîne. On insère ainsi un caractère anti-slash « \ » avant les apostrophes contenues dans le message.



On doit également échapper les guillemets si on utilise les guillemets comme délimiteurs.

Le caractère d'échappement « \ » est utilisé pour créer d'autres signes très utiles. Ainsi, « \n » symbolise un saut de ligne ("essai\nsur\nplusieurs\nlignes"). Pour écrire un véritable anti-slash dans une chaîne, il faut l'échapper lui-même (et donc écrire « \\ »).

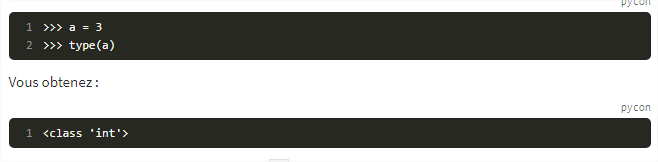


La fonction « type »

La fonction renvoie le type de la variable passée en paramètre.

type(nom\_de\_la\_variable)

Python vous indique donc que la variable a appartient à la classe des entiers.



Str est l'abréviation de « string » qui signifie chaîne (sous-entendu, de caractères) en anglais.

Les structures conditionnelles

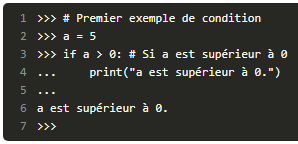
Les conditions permettent d'exécuter une ou plusieurs instructions dans un cas, d'autres instructions dans un autre cas.

En Python, un commentaire débute par un dièse (« # ») et se termine par un saut de ligne.

On entend par indentation un certain décalage vers la droite, obtenu par un (ou plusieurs) espaces ou tabulations.

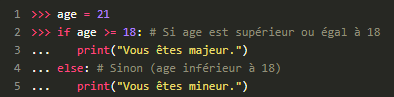
Les indentations sont essentielles pour Python. Il ne s'agit pas, comme dans d'autres langages tels que le C++ ou le Java, d'un confort de lecture mais bien d'un moyen pour l'interpréteur de savoir où se trouvent le début et la fin d'un bloc.

Exemple d'un if en python



Après la condition il ne faut pas oublier les deux points :

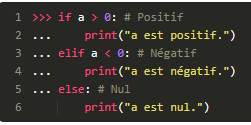
**L'instruction else:**



On oublie pas les 2 points après un else:

L'instruction elif:

Le mot clé elif est une contraction de « else if », que l'on peut traduire très littéralement par « sinon si », on l'utilise si on a plusieurs conditions à vérifier



L'instruction else ne peut figurer qu'une fois contrairement à elif, clôturant le bloc de la condition. Deux instructions else dans une même condition ne sont pas envisageables et n'auraient de toute façon aucun sens.

Attention : l'égalité de deux valeurs est comparée avec l'opérateur « == » et non « = ». Ce dernier est en effet l'opérateur d'affectation et ne doit pas être utilisé dans une condition.

N'oubliez pas que True et False sont des valeurs ayant leur première lettre en majuscule. Si vous commencez à écrireTrue sans un 'T' majuscule, Python ne va pas comprendre.

Les opérateurs logiques :

Ou >> or

Et >> and

Non >> not

Avec un or la condition ets vraie si l'une des deux est vraies

>>> a, b = 4, 5 *#a recoit 4 et b recoit 5. Assez cool non :)?*

2 >>> a == 4 **or** b == 9 *#Vu que l'une des conditions est vraie on a donc True*

3 True

Avec un and, la condition est vraie si les deux sont vraies

1 >>> a, b, c = 4, 5, 9 *#a recoit 4 et b recoit 5, c recoit 9.*

2 >>> a == 4 **and** b == 5 **and** c == 6 *#Vu que la dernière condition est fausse on a d\*

3 onc False

4 False

5 >>>

Not retourne True si vous lui donnez False et False si vous lui donner True.

est\_malade = True

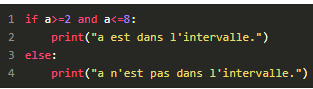
2

3 **if**(**not** est\_malade):

4 **print**("Vous n’êtes pas malade.")

5 **else**:

6 **print**("Vous êtes malade!")



Programme afin de savoir si une année est bissextile ou pas, rappel

Une année est dite bissextile si c'est un multiple de 4, sauf si c'est un multiple de 100. Toutefois, elle est considérée comme bissextile si c'est un multiple de 400. Je développe :

Si une année n'est pas multiple de 4, on s'arrête là, elle n'est pas bissextile.

Si elle est multiple de 4, on regarde si elle est multiple de 100.

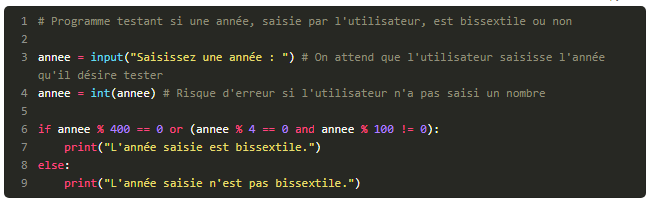
Si c'est le cas, on regarde si elle est multiple de 400.

Si c'est le cas, l'année est bissextile.

Sinon, elle n'est pas bissextile car multiple de 4 et de 100 mais pas de 400

Sinon, elle est bissextile car multiple de 4 mais pas de 100.

Correction :



Les boucles

La boucle while

Syntaxe:

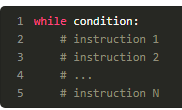


Table de multiplication de 7

nb = 7 # On garde la variable contenant le nombre dont on veut la table de multiplication

i = 1 # C'est notre variable compteur que nous allons incrémenter dans la boucle

while i <= 10: # Tant que i est inférieure ou égale à 10

print(i, "\*", nb, "=", i \* nb)

i += 1 # On incrémente i de 1 à chaque tour de boucle

La boucle for

Elle est diffrénte de celle dans les autres langages, elle ressemble au foreach en PHP.

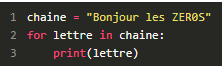
L'instruction for travaille sur des séquences. Elle est en fait spécialisée dans le parcours d'une séquence de plusieurs données.

La boucle Python for parcourt les items d’une séquence (liste, chaîne, etc.), dans l’ordre d’apparition dans la séquence.

Syntaxe:



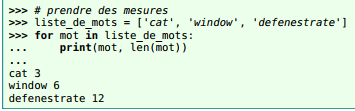
element est une variable créée par le for, ce n'est pas à vous de l'instancier. Elle prend successivement chacune des valeurs figurant dans la séquence parcourue.



Résultat:



Notez bien que, du coup, il est inutile d'incrémenter la variable lettre (ce qui serait d'ailleurs assez ridicule vu que ce n'est pas un nombre). Python se charge de l'incrémentation, c'est l'un des grands avantages de l'instruction for.



Pour afficher les mots sur une seule ligne met dans ton print end ="" , ex:



Va afficher >> Torcy



**For i in range**

La fonction range() permet de retourner une séquence virtuelle de nombres de début à fin par pas de pas.

Le début et le pas par défaut sont respectivement 0 et 1.

Le premier paramètre est inclus, le second quant à lui est exclus.

for i in range(5):

print(i)

Va afficher:

0

1

2

3

4

for i in range(1, 11):

print(i)

Va afficher:

1

2

3

4

5

6

7

8

9

10

Pour les afficher sur une seule ligne tu fais print(i, end= '' '')

Pour modifier le pas on l'indique en troisième chiffre, ex:

for i in range(0, 11, 2):

print(i, end= '' '')

Va afficher: 0 2 4 6 8 10

Le mot-clé **in** peut être utilisé ailleurs que dans une boucle for.

1 chaine= "Hello World"

2 **for** letter **in** chaine:

3 **if** letter **in** "aeiouyAEIOUY":

4 **print**(letter)

Cela va afficher la lettre si c'est une voyelle, donc là on aura:

e

o

o

>>>

Un petit bonus : les mots-clés break et continue

Le mot-clé break

Ce mot-clé permet d'arrêter la boucle appelante quelle que soit la condition de la boucle. Python sort immédiatement de la boucle et exécute le code qui suit la boucle, s'il y en a.

**for** i **in** range(0, 15):

**if**(i == 2):

**break**

**print**(i)

Ce programme affichera seulement 0, 1 étant donné que lorsque i vaudra 2 la boucle s’arrêtera automatiquement à cause du fameux break.

Double boucle :

Voici un programme:



Il va afficher:

0 10

0 11

0 12

1 10

1 11

1 12

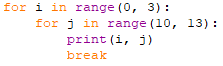
2 10

2 11

2 12

Avec un break:

0 10



1 10

2 10

Le break arrête la boucle appelante et non toutes les boucles. Ainsi notre fameux break arrêtera simplement la seconde boucle.

Si on met le break dans la première boucle:

Il affichera:



0 10

0 11

0 12

Voyons en détails ce qui passe pour que tout soit clair:

Lors du premier tour de boucle i vaudra 0. On rentre ensuite dans la seconde boucle j vaut 10. Ainsi on affiche 0 10. Ensuite on rencontre le break, ce qui fait que le j ne passera pas à 11. On sort donc de la seconde boucle et on retourne à la première boucle.

A présent i vaut 1, on rentre dans la seconde boucle j vaut 10. Ainsi on affiche 1 10. Ensuite on rencontre le break, ce qui fait que le j ne passera pas également à 11. On sort donc de la seconde boucle et retourne à la première boucle.

Le i vaut maintenant 2 et ainsi de suite…

Use of break statement inside loop  
for val in "string":  
 if val == "i":  
 break  
 print(val)  
print("The end")

Output:

s

t

r

The end

Le mot-clé continue

Il permet de passer simplement au prochain tour de boucle.

1 #Affiche les nombres pairs de 0 à 15

2 for i in range(0, 15):

3 if(i % 2 != 0):

4 continue

5 print(i)

Le mot-clé continue comme on vient de le voir n’arrête pas la boucle mais permet de passer simplement au prochain tour de boucle.

Pas à pas vers la modularité (1/2)

Les modules permettent de regrouper plusieurs fonctions selon le même principe. Toutes les fonctions mathématiques, par exemple, peuvent être placées dans un module dédié aux mathématiques.

**La création de fonctions**

Les fonctions permettent de regrouper plusieurs instructions.

Par exemple si tu veux écrire la table de multiplication de 5 tu lui diras:

Etape 1

Etape 2

Etape 3, etc..

Tu vas regrouper toutes ces étapes dans une fonction appelée ainsi

def table\_de\_5():

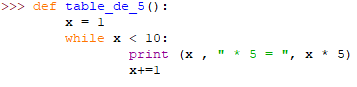
Instructions...

Syntaxe:

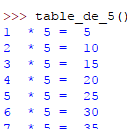


Exemple d'une fonction de la table de multiplication par 5

Ensuite pour l'afficher tu tapes juste ça:



>>> table\_5()



**Fonction avec un paramètre**

def passer\_le\_salam(nom):

print(“ Salam mon petit “ + nom)

>>>salam(“Ilyas”)

Salam mon petit Ilyas

Lorsqu’on définit une fonction, les ingrédients ajoutés entre parenthèses sont appelés paramètres. Par contre, lors de l’appel de la fonction, on parle plutôt d’arguments.

**Fonction avec un paramètre ayant une valeur par défaut**

def bonjour( nom = “boss”):

print (“ Bonjour “ + nom)

>>>bonjour()

Bonjour boss

>>>bonjour(“Sarah”)

bonjour Sarah

La signature d’une fonction est tout simplement le nom de la fonction. Cela signifie que vous ne pouvez pas définir deux fonctions ayant le même nom. Si vous le faites, l’ancienne définition est écrasée par la nouvelle.

Lorsque vous utilisez la fonction print en séparant les éléments par une virgule, elle se chargera d’afficher vos éléments séparés par un espace. De plus aucun casting n’est requis.

**print**("Salut", 2) *#Aucune erreur! Affichera Salut 2*

**L'instruction return**

Print affiche mais ne renvoie rien, type et input renvoient une valeur

Une fonction peut être en effet utilisée pour retourner une valeur.

1 **def** est\_pair(nombre):

2 *"""Renvoie True si nombre est pair*

3 *et False dans le cas contraire"""*

4

5 **if**(nombre % 2 == 0):

6 **return** True

7 **else**:

8 **return** False



La variable variable contiendra, après exécution de cette instruction, 5 au carré, c'est-à-dire 25.

Max est une fonction de python qui retourne le plus grand nombre

1 **def** maximum(nbre1 = 0, nbre2 = 100):

2 **return** max(nbre1, nbre2)

1 maximum() *#retournera 100*

2 maximum(40, 300) *#Retournera 300*

Dès que l’interpréteur exécute l’instruction **return**, tout ce qui vient après ne sera pas exécuté.

Ainsi dans l’exemple qui suit, Je suis cool ne sera donc pas affiché, seule la chaîne Je serai affichée sera affichée

1 **def** petit\_test():

2 **print**("Je serai affichée")

3 **return** True

4 **print**("Je suis cool")

Les fonctions lambda

Un autre moyen de créer des fonctions, sans le mot clé def, ce sont des fonctions extrêmement courtes car limitées à une seule instruction.

Syntaxe:

lambda arg1, arg2,… : instruction de retour.

Exemple d'une fonction lamba ayant qu'un seul argument :

lambda b: x\*x

Pour pouvoir l'utiliser il faut la mettre dans une variable, ex:

>>>f = lambda b: x \* x

print(f(5))

25

Un autre exemple : si vous voulez créer une fonction lambda prenant deux paramètres et renvoyant la somme de ces deux paramètres, la syntaxe sera la suivante :

lambda x, y: x + y

f(2,5)

print(f(2,5))

7

À la découverte des modules

**Les modules, qu'est-ce que c'est ?**

Un module est grossièrement un bout de code que l'on a enfermé dans un fichier

On emprisonne ainsi des fonctions et des variables ayant toutes un rapport entre elles. Ainsi, si l'on veut travailler avec les fonctionnalités prévues par le module (celles qui ont été enfermées dans le module), il n'y a qu'à **importer** le module et utiliser ensuite toutes les fonctions et variables prévues.

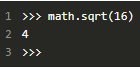
Pour utiliser un modul l il faut l'importer, syntaxe:

import nomDuModule

Ex:



Pour appeler une fonction du module, il faut taper le nom du module suivi d'un point « . » puis du nom de la fonction. C'est la même syntaxe pour appeler des variables du module. Voyons un exemple :



Mais comment suis-je censé savoir quelles fonctions existent et ce que fait math.sqrt dans ce cas précis ?

J'aurais dû vous montrer cette fonction bien plus tôt car, oui, c'est une fonction qui va nous donner la solution. Il s'agit de help, exemple:

help(“math”) ou help(“math.sqrt”)

Tapez Q pour revenir à la fenêtre d'interpréteur.

Une autre méthode d'importation :from … import …

Admettons que nous ayons uniquement besoin, dans notre programme, de la fonction renvoyant la valeur absolue d'une variable. Dans ce cas, nous n'allons importer que la fonction, au lieu d'importer tout le module.

>>> from math import fabs

>>> fabs(-5)

5

>>> fabs(2)

2

>>>

Pas à pas vers la modularité (2/2)

Nous allons commencer par voir comment mettre nos programmes en boîte

On peut enregistrer ses scripts dans des fichiers ayant l'extension.py

On peut ainsi créer ses propres modules

Les packages

un package sert à regrouper plusieurs modules.

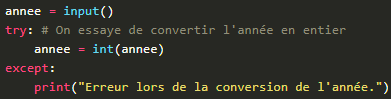
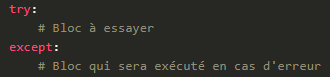
**En pratique**

En pratique, les packages sont… des répertoires ! Dedans peuvent se trouver d'autres répertoires (d'autres packages) ou des fichiers (des modules).

Les exceptions

Forme minimale du bloc try

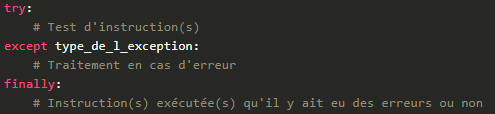
On va parler ici de bloctry. Nous allons en effet mettre les instructions que nous souhaitons tester dans un premier bloc et les instructions à exécuter en cas d'erreur dans un autre bloc. Sans plus attendre, voici la syntaxe :



Si tu mets des lettres tu auras le message du print.

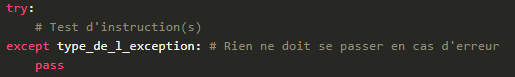
Le mot-clé finally

finally permet d'exécuter du code après un bloctry, quel que soit le résultat de l'exécution dudit bloc. La syntaxe est des plus simples :



Un petit bonus : le mot-clé pass

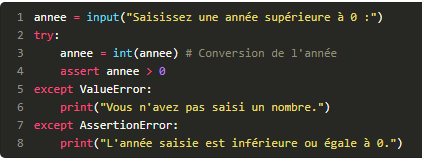
Il peut arriver, dans certains cas, que l'on souhaite tester un bloc d'instructions… mais ne rien faire en cas d'erreur. Toutefois, un bloctryne peut être seul.



Les assertions

Les assertions sont un moyen simple de s'assurer, avant de continuer, qu'une condition est respectée. En général, on les utilise dans des blocs try … except.

Dans le programme testant si une année est bissextile, on pourrait vouloir s'assurer que l'utilisateur ne saisit pas une année inférieure ou égale à 0 par exemple. Avec les assertions, c'est très facile à faire :



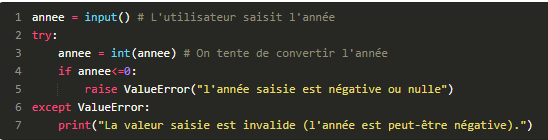
Lever une exception

Avec le mot **raise**

Syntaxe:



Prenons un petit exemple, toujours autour de notre programme bissextile. Nous allons lever une exception de type ValueErrorsi l'utilisateur saisit une année négative ou nulle.



En résumé

On peut intercepter les erreurs (ou exceptions) levées par notre code grâce aux blocs try except.

La syntaxe d'une assertion est assert test.

Les assertions lèvent une exception AssertionError si le test échoue.

On peut lever une exception grâce au mot-clé raise suivi du type de l'exception.

Les chaînes de caractères

chaine = str() # Crée une chaîne vide

# On aurait obtenu le même résultat en tapant chaine = ""

while chaine.lower() != "q":

print("Tapez 'Q' pour quitter...")

chaine = input()

print("Merci !")

.lower >> va mettre en minuscule

.upper >> va mettre en majuscule

**Méthode format**

**>>> prenom = "Paul"**

**>>> nom = "Dupont"**

**>>> age = 21**

**>>> print("Je m'appelle {0} {1} et j'ai {2} ans.".format(prenom, nom, age))**

**Je m'appelle Paul Dupont et j'ai 21 ans.**

Nous avons utilisé une méthode de la classe str pour formater notre chaîne. De gauche à droite, nous avons :

une chaîne de caractères qui ne présente rien de particulier, sauf ces accolades entourant des nombres, d'abord 0, puis 1, puis 2 ;

nous appelons la méthode format de cette chaîne en lui passant en paramètres les variables à afficher, dans un ordre bien précis ;

quand Python exécute cette méthode, il remplace dans notre chaîne {0} par la première variable passée à la méthode format (soit le prénom), {1} par la deuxième variable… et ainsi de suite.

On aurait pu écrire cela ainsi effectivement :

prenom = "Paul"

nom = "Dupont"

age = 21

print("Je m'appelle " + prenom + " " + nom + " et j'ai ", age, " age .")

ou ainsi:

prenom = "Paul"

nom = "Dupont"

age = 21

print("Je m'appelle " + prenom + " " + nom + " et j'ai " + str(age) + " age .")

On appelle str pour convertir un objet en une chaîne de caractères, comme nous avons appelé int pour convertir un objet en entier. C'est le même mécanisme, sauf que convertir un entier en chaîne de caractères ne lèvera vraisemblablement aucune exception.

On peut la mettre dans une variable:

>>> nouvelle\_chaine = "Je m'appelle {0} {1} et j'ai {2} ans.".format(prenom, nom, age)

Parcours et sélection de chaînes

**Parcours par indice**

>>> chaine = "Salut Sami "

>>> chaine[0] # Première lettre de la chaîne

'S'

>>> chaine[2] # Troisième lettre de la chaîne

'l'

>>> chaine[-1] # Dernière lettre de la chaîne

'i'

>>>

Quand vous tapez chaine[-1], vous accédez ainsi à la dernière lettre de la chaîne (enfin, au dernier caractère, qui n'est pas une lettre ici).

On peut obtenir la longueur de la chaîne (le nombre de caractères qu'elle contient) grâce à la fonction len.

>>> chaine = "Salut"

>>> len(chaine)

5

>>>

Méthode de parcours par while

chaine = "Salut"

i = 0 # On appelle l'indice 'i' par convention

while i < len(chaine):

print(chaine[i]) # On affiche le caractère à chaque tour de boucle

i += 1

**Sélection de chaînes**

>>> presentation = "salut"

>>> presentation[0:2] # On sélectionne les deux premières lettres

'sa'

>>> presentation[2:len(presentation)] # On sélectionne la chaîne sauf les deux premières lettres

'lut'

>>>

>>> presentation[:2] # Du début jusqu'à la troisième lettre non comprise

'sa'

>>> presentation[2:] # De la troisième lettre (comprise) à la fin

'lut'

>>>

Les tuples

Les tuples vous permettent de stocker une liste de valeurs qui ne changeront pas durant tout le programme.

Syntaxe: nom\_du\_Tuple = (''valeur1'', ''valeur2'', ''valeur3'')²

mois = ("Janvier", "Février", "Mars", "Avril", "Mai", "Juin", "Juillet", "Août", "Septembre", "Octobre", "Novembre", "Décembre")

Tu ne peux ajouter un élément à un tuple avec append, tu es obligé de réécrire tout le tuple.

**Les listes.**

Les listes sont une liste de valeurs. A la différence des tuples, vous pouvez ajouter, modifier et supprimer des valeurs de la liste.

villes = ["Torcy", "Lognes", "Lyon"]

Création d'une liste vide:

Pays = [ ]

Les listes peuvent contenir n'importe quel type d'objet.

liste = [1, 2, 3, "ludovic"]

**Ajouter un élément à la fin de la liste**

On utilise la méthode append pour ajouter un élément à la fin d'une liste.

villes = ["Torcy", "Lognes", "Lyon"]

villes.append ("Marseille")

>>>villes

["Torcy", "Lognes", "Lyon", "Marseille"]

**Insérer un élément dans la liste**

On peut, très simplement, insérer un objet dans une liste, à l'endroit voulu. On utilise pour cela la méthode insert.

villes.insert("Bussy", 2)

Tu demandes à insérer Bussy à l'indice 2 donc:

>>>print(villes)

["Torcy", « Lognes","Bussy", "Lyon", "Marseille"]

**Concaténation de listes**

liste1 = [1, 2, 3]

liste2 = [4, 5, 6]

1ère méthode :

>>>liste1 + liste2

[1, 2, 3, 4, 5, 6]

2nd méthode :

liste1.extend(liste2)

>>>print(liste1)

[1, 2, 3, 4, 5, 6]

**Suppression d'éléments d'une liste**

On utilise les mots del et remove

La méthode del

age = 21

del age

La variable age n'existe plus.

Villes = [''Torcy'', ''Lognes'', ''Bussy'', ''Lyon'', ''Marseille'']

del Villes[3]

>>>Villes

[''Torcy'', ''Lognes'', ''Bussy'', ''Marseille'']

La méthode remove

On peut aussi supprimer des éléments de la liste grâce à la méthode remove qui prend en paramètre non pas l'indice de l'élément à supprimer, mais l'élément lui-même.

Villes.remove(''Marseille'')

>>>Villes

[''Torcy'', ''Lognes'', ''Bussy'']

La méthode remove ne retire que la première occurrence de la valeur trouvée dans la liste !

Connaître la longueur d'une liste

On utilise le mot len

Pays = ["Japon", "Russie", "Jordanie", "Sénégal", "Indonésie"]

print (len(Pays))

5

**Parcourir une liste**

>>> ma\_liste = ['a', 'b', 'c']

>>> i = 0 # Notre indice pour la boucle while

>>> while i < len(ma\_liste):

... print(ma\_liste[i])

... i += 1 # On incrémente i, ne pas oublier !

...

a

b

c

>>> # Cette méthode est cependant préférable

... for elt in ma\_liste: # elt va prendre les valeurs successives des éléments de ma\_liste

... print(elt)

a

b

c

>>>

**La fonction enumerate**

**Enumerate prend en paramètre une liste et renvoie l'indice et sa valeur dans la liste.**

>>> ma\_liste = ['a', 'b', 'c']

**>>> for elt in enumerate(ma\_liste):**

**... print(elt)**

**...**

**(0, 'a')**

**(1, 'b')**

**(2, 'c')**

**Tu peux mettre x à la place de elt**

**Si tu mets print(ma\_liste) tu auras:**

['a', 'b', 'c']

['a', 'b', 'c']

['a', 'b', 'c']

>>> ma\_liste = ['a', 'b', 'c']

>>> for i, elt in enumerate(ma\_liste):

... print("À l'indice {} se trouve {}.".format(i, elt))

...

À l'indice 0 se trouve a.

À l'indice 1 se trouve b.

À l'indice 2 se trouve c.

>>>

Quand on utilise enumerate, on capture l'indice et l'élément dans deux variables distinctes.

**Les compréhensions de liste**

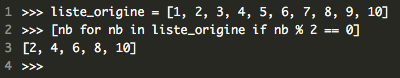
Les **compréhensions de liste** permettent de parcourir une liste en en renvoyant une seconde, modifiée ou filtrée.

>>> liste\_origine = [0, 1, 2, 3, 4, 5]

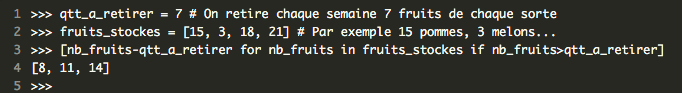
>>> [nb \* nb for nb in liste\_origine]

[0, 1, 4, 9, 16, 25]

>>>

**Filtrage avec un branchement conditionnel**

On rajoute à la fin de l'instruction une condition qui va déterminer quelles valeurs seront transférées dans la nouvelle liste. Ici, on ne transfère que les valeurs paires



**La portée des variables**

La portée utilisée dans ce sens c'est « quand et comment les variables sont-elles accessibles ? ».

L'**espace local** d’une fonction contient les paramètres qui sont passés à la fonction et les variables définies dans son corps.

Les variables définies à l’intérieur de la fonction ne sont accessibles que dans la fonction et non en dehors.

Python a une règle d'accès spécifique aux variables extérieures à l'espace local : on peut les lire, mais pas les modifier.

une fonction ne peut modifier, par affectation, la valeur d'une variable extérieure à son espace local.

**Une fonction modifiant des objets**

J'espère que vous vous en souvenez, en Python, tout est objet. Quand vous passez des paramètres à votre fonction, ce sont des objets qui sont transmis. Et pas les valeurs des objets, mais bien les objets eux-mêmes, ceci est très important.

>>> def ajouter(liste, valeur\_a\_ajouter):

... """Cette fonction insère à la fin de la liste la valeur que l'on veut ajouter"""

... liste.append(valeur\_a\_ajouter)

...

>>> ma\_liste=['a', 'e', 'i']

>>> ajouter(ma\_liste, 'o')

>>> ma\_liste

['a', 'e', 'i', 'o']

>>>

Ça fonctionne car on ne change pas du tout la valeur du paramètre, on appelle juste une méthode de l'objet.

Les références :

les variables que nous utilisons depuis le début de ce cours cachent en fait des références vers des objets.

Une variables est un nom identifiant pointant vers une valeur.

En fait, une variable est un nom identifiant, pointant vers une référence d'un objet. La référence, c'est un peu sa position en mémoire. Cela reste plus haut niveau que les pointeurs en C par exemple, ce n'est pas vraiment la mémoire de votre ordinateur. Et on ne manipule pas ces références directement.

Cela signifie que deux variables peuvent pointer sur le même objet.

>>> ma\_liste1 = [1, 2, 3]

>>> ma\_liste2 = ma\_liste1

>>> ma\_liste2.append(4)

>>> print(ma\_liste2)

[1, 2, 3, 4]

>>> print(ma\_liste1)

[1, 2, 3, 4]

>>>

Nous créons une liste dans la variablema\_liste1. À la ligne 2, nous affectonsma\_liste1à la variablema\_liste2. On pourrait croire quema\_liste2est une copie dema\_liste1. Toutefois, quand on ajoute4àma\_liste2,ma\_liste1est aussi modifiée.

On dit quema\_liste1etma\_liste2contiennent une référence vers le même objet : si on modifie l'objet depuis une des deux variables, le changement sera visible depuis les deux variables.

Ceci ne fonctionne pas avec des variables contenant des entiers, flottants ou string car elles n’ont pas de méthodes travaillant sur l’objet lui même.

**Les variables globales**

Il existe un moyen de modifier, dans une fonction, des variables extérieures à celle-ci. On utilise pour cela des **variables globales**.

**Le principe des variables globales**

On ne peut faire plus simple. On déclare dans le corps de notre programme, donc en dehors de tout corps de fonction, une variable, tout ce qu'il y a de plus normal.Dans le corps d'une fonction qui doit modifier cette variable (changer sa valeur par affectation), on déclare à Python que la variable qui doit être utilisée dans ce corps est globale.

>>> i = 4 # Une variable, nommée i, contenant un entier

>>> def inc\_i():

... """Fonction chargée d'incrémenter i de 1"""

... global i # Python recherche i en dehors de l'espace local de la fonction

... i += 1

...

>>> i

4

>>> inc\_i()

>>> i

5

>>>

En précisantglobal i, Python permet l'accès en lecture et en écriture à cette variable, ce qui signifie que vous pouvez changer sa valeur par affectation.