

Initiation à la programmation avec Python

Jean-Pierre Messenger (jp@xiasma.fr)

7 juillet 2020 – version 1.2a



Utilisation commerciale interdite sans autorisation

Conditions de distribution

Licence Creative Commons

Attribution - Pas d'Utilisation Commerciale

Pas de Modification 3.0 France

(CC BY-NC-ND 3.0 FR)

Ceci est un résumé de la licence complète disponible à :

<https://creativecommons.org/licenses/by-nc-nd/3.0/fr/legalcode>

Vous êtes autorisé à :

Partager — copier, distribuer et communiquer le matériel par tous moyens et sous tous formats.

L'offrant ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence.

Selon les conditions suivantes :

Attribution — Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que l'offrant vous soutient ou soutient la façon dont vous avez utilisé son œuvre.

Pas d'utilisation commerciale — Vous n'êtes pas autorisé à faire un usage commercial de cette œuvre, tout ou partie du matériel la composant. Le titulaire des droits peut autoriser tous les types d'utilisation ou au contraire restreindre aux utilisations non commerciales (*les utilisations commerciales restent soumises à son autorisation.*)

Pas de modifications — Dans le cas où vous reprenez ce document dans une autre œuvre, que vous transformez, ou créez à partir du matériel composant l'œuvre originale, vous n'êtes pas autorisé à distribuer ou mettre à disposition l'œuvre modifiée.

Pas de restrictions complémentaires — Vous n'êtes pas autorisé à appliquer des conditions légales ou des mesures techniques qui restreindraient légalement autrui à utiliser l'œuvre dans les conditions décrites par la licence.

Télécharger ce cours et ses exercices

Il est disponible sur un dépôt git

Si vous le souhaitez...

- Créez votre compte sur <https://framagit.org/>
- Transmettez-moi votre identifiant
- Je vous accorde le status *reporter* sur ce projet
 - Accès en lecture
 - Ouverture de tickets
- Il est là : <https://framagit.org/jpython/introduction-programmation-en-python>
- D'autres projets : <https://framagit.org/jpython/meta>
- Des mises à jours sont régulièrement disponibles
- Chaque changement de version est accompagné d'un *tag*

Initiation à la programmation avec Python

Plan du cours

- Système, programmes et langages
- Notre premier programme
- Accès et traitement des données
- Données composites
- Boucles
- Fonctions
- Maintenance, débogage et tests
- Programmation Orientée Objet

Systèmes, programmes et langages

- Architecture des ordinateurs et systèmes d'exploitation
- Qu'est-ce qu'un langage de programmation ?
- Algorithmes
- Paradigmes des langages de programmations
- Compilateurs et interpréteurs
- Exemples en divers langages

Notre premier programme

- Syntaxe, expressions et instructions
- Compilation et exécution
- Bibliothèques de programmation
- Bonnes pratiques

Accès et traitement des données

- Noms et références
- Types de données
- Opérateurs, fonctions et méthodes
- Booléens et alternatives

Données composites

- Collections de données
- Listes
- Modification et extraction d'informations
- Autres collections

Boucles

- Parcours de listes
- Parcours des autres collections
- Compréhensions

Fonctions

- Structuration du code avec des fonctions
- Passage de paramètres
- Renvoi de valeur
- Appels de fonction

Maintenance, débogage et tests

- Interpréter les messages d'erreur
- Utilisation d'un débogueur pas à pas
- Test unitaires

Programmation Orientée Objet

- Classes et instances
- Attributs et méthodes
- Héritage
- Méthodes spéciales

Architecture d'un ordinateur

- CPU *Central Processing Unit* : unité de calcul
- Mémoire centrale (RAM)
 - Le CPU manipule des données réalise des calculs sur elles
 - Il transfère des données vers et à partir de la mémoire
 - Les programmes sont stockés en RAM eux aussi
 - La mémoire n'est pas permanente
- Bus (PCI, etc.) pour dialoguer avec les périphériques
 - Clavier, écran, espaces de stockage (disques, SSD, clé USB, ...)
 - Cartes réseau
 - Cartes son
 - ...

Systèmes d'exploitation

- On peut programmer directement un ordinateur « vide » : sur le *métal* (se rencontre parfois dans l'informatique embarquée)
- La plupart des ordinateurs démarrent en lançant un programme particulier le *noyau* du système d'exploitation
 - Le noyau est un logiciel qui accède au matériel réellement
 - Il isole les programmes les uns des autres
 - Il les fait tourner « en même temps » en les interrompant régulièrement
 - Il fournit des services variés
 - Accès au stockage à travers de fichiers
 - Communication réseau
 - Accès au matériel en général (son, écran, clavier)
 - Il est accompagné d'utilitaires et d'outils divers
- Exemples : GNU/Linux, Android (Linux), UNIX (Solaris, macOS X, AIX, ...), VMS, zOS, ReactOS, Plan 9, Microsoft Singularity & Midori
- Autres exemples : MS-DOS, Microsoft Windows

Langages de programmation

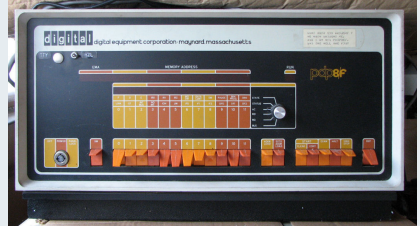
Vous pouvez écrire directement un programme pour un CPU en binaire.

Binaire

```
110111001011101111000100110101011110011011110111110000
100110010100111010010101101101011111000110011101011011
111100111011111011111100000100001100010100011100100100
101100110100111101000101001101010101011101100101101101
110101111110000110001110010110011110100110101110110110
11111100011100111101011101111110011110111111011111100
000010000011000010100001110001001000101100011010001111
001000100100110010101001011100110010011011001110100111
110100001010001101001010100111010100101010110101101010
111101100010110011011010101101110111001011101101111010
111111100000110000111000101100011110010011001011100110
0100000101000011010001010100011101...
```

Comment programmer en binaire ?

- Haut = 0, bas = 1
- Donc : 100011101101
- 2285 en base 10
- 8ED en base 16 (chiffres : 0, 1, 2, ... 9, A, B, C, D, E, F)
- Peut vouloir dire :
 - Ajoute 1 au registre A1
 - Copie l'adresse mémoire 8E en registre A7
 - Mets la valeur 142 en registre A2
 - Copie 142 à l'adresse RAM 13
 - On en sait rien *a priori*
 - C'est dépendant du processeur et du contexte



Langage machine

- Une collection de termes « mnémoniques » associés à des nombres
- Toujours dépendant de l'architecture du processeur

Un peu d'assembleur x86_64 (PC, Mac, ...)

```
movq    -32(%rbp), %rax
addq    $8, %rax
movq    (%rax), %rax
movzbl  (%rax), %eax
movzbl  -1(%rbp), %eax
addl    $1, %eax
leaq    -3(%rbp), %rax
```

- On peut convertir ensuite ce texte en binaire à la main ou en utilisant un programme d'*assemblage*

Langages de programmation

- Un langage plus simple à écrire, plus lisible, plus concis qui sera converti en programme binaire
- Peut être *compilé* (traduit en langage machine) ou *interprété* (exécuté sous le contrôle d'un autre programme)
- Pour en tester quelques (!) uns : <http://tio.run/>

Un programme en langage Pascal :

```
Program myprogram;  
Var  
    n: integer;  
Begin  
    Write('Nombre : ');  
    Readln(n);  
    n := n + 42;  
    Writeln(n);  
End.
```

Algorithmes

- Concept mathématique qui date de l'antiquité (algorithme d'Euclide, d'Érathostène, etc.)
- Étymologie : du mathématicien Al-Khwârizmî, Perse, IX^e siècle
- Description détaillée, *mécanique*, d'un *calcul*

Algorithme d'Érathostène

- * Écrire les nombres de 2 à N dans des cases.
- * Commencer avec $k=2$
- * Tant que $k \times k < N$:
 - Trouver le premier nombre non barré non déjà examiné, soit k
 - Barrer un nombre sur k dans la suite sauf la case où est k
- * Renvoyer les nombres dans les cases non barrées

On obtient les nombres premiers de 1 à N

L'algorithme d'Ératosthène en action

Étapes pour $N = 25$

- Liste de départ :

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

- On prend le premier nombre non barré $k=2$, et on barre une case sur deux :

2 3 4 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 14 15 ~~16~~ 17 ~~18~~ 19 20 21 ~~22~~ 23 24 25

- $2 \times 2 < 25$? Oui, on continue, $k=3$:

2 3 4 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 14 ~~15~~ ~~16~~ 17 ~~18~~ 19 20 21 ~~22~~ 23 24 25

- $3 \times 3 < 25$? Oui, on continue, $k=5$:

2 3 4 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 14 ~~15~~ ~~16~~ 17 ~~18~~ 19 20 21 ~~22~~ 23 24 25

- $5 \times 5 < 25$? Non, on arrête.

Résultat : 2 3 5 7 11 13 17 19 23

Algorithmes

- Un algorithme doit être décrit de façon non ambiguë
- Il est applicable *mécaniquement*
 - Nul besoin de *comprendre* de quoi il s'agit
 - Une machine pourrait le faire, comme un métier à tisser
- Les premières machines de calculs étaient mécaniques (calculatrices de Pascal et de Leibniz, machine de Charles Babbage, caisses enregistreuses)
- L'électronique a permis d'en construire de bien plus rapides et pouvant enregistrer plus d'information
- À partir d'un certain niveau de complexité elles sont toutes *équivalentes* :
 - Peuvent résoudre exactement la même classe de problèmes
 - C'est une chose qui est mathématiquement prouvée

Définition de la factorielle

Définition mathématique

$$0! = 1$$

$$n! = \prod_{k=1}^n k = 1 \times 2 \times 3 \times \dots \times n \quad \text{si } n > 0$$

Exemples

$$0! = 1$$

$$1! = 1$$

$$2! = 2$$

$$3! = 6$$

$$4! = 240$$

$$5! = 120$$

$$6! = 720$$

$$7! = 5040$$

52! est le nombre de façons d'organiser un jeu de cartes :

80658175170943878571660636856403766975289505440883
277824000000000000

Algorithmes : factorielle

Premier algorithme (itératif)

```
Si n = 0 renvoyer 1
Sinon :
    Partir de i = 1 et res = 1
    Tant que i est inférieur ou égal à n :
        Stocker res × i dans res
        Augmenter i de 1
    Renvoyer res
```

Second algorithme (récursif)

```
Si n = 0 renvoyer 1
Sinon renvoyer n × le résultat obtenu pour n - 1
```

Factorielle : Pascal (itératif)

Pascal itératif

```
Program factorielle;
Var
    n, i, res: integer;
Begin
    Write('Nombre : ');
    ReadLn(n);
    WriteLn();
    res := 1;
    { Ne fait rien si i > n }
    for i := 1 to n do
        res := res * i;
    End;
    WriteLn(res);
End.
```

Factorielle : C

C itératif

```
#include<stdio.h>
#include<stdlib.h>

int main(void) {
    int n, i, res=1;
    printf("Nombre : ");
    scanf("%d", &n);
    /* Ne fait rien si i > n */
    for (i = 1; i <= n; i++) {
        res *= i;
    }
    printf("%d\n", res);
    exit(0);
}
```

Factorielle : Scheme

Scheme récursif simple

```
(define factorial
  (lambda (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1)))))
```

Scheme récursif optimisé (récursivité terminale)

```
(define factorial-tail
  (lambda (n acc)
    (if (= n 0)
        acc
        (factorial-tail (- n 1) (* acc n)))))
```

Factorielle : Java

Java itératif

```
class Factorial{  
    public static void main(String args[]){  
        int i, res = 1;  
        int n = Integer.parseInt(args[0]);  
        for(i = 1; i <= number; i++){  
            res = res * i;  
        }  
        System.out.println(  
            "La factorielle de " + number +  
            " est : " + res  
        );  
    }  
}
```

Factorielle : OCaml, Haskell

OCaml récursif

```
let rec factorielle n =  
  if n<2 then 1 else n*factorielle (n-1);;
```

Haskell récursif

```
fac n = if n > 0 then n * fac(n - 1) else 1
```

Haskell itératif

```
fac n = product [1..n]
```

Factorielle : Python

Python itératif

```
n = int(input('Nombre :'))
res = 1
# Boucle de 1 à n
for i in range(1, n+1):
    res = res * i
print(res)
```

Bien d'autres exemples d'implémentations sur
<https://rosettacode.org/wiki/Factorial>

Une frise sur l'histoire des langages de programmation :
<https://www.levenez.com/lang/>

Paradigmes des langages de programmation

Différents *paradigmes* logiques

- Impératif : suite d'instructions et d'expressions
 - C, Pascal, Fortran
- Fonctionnel : appels et manipulations de fonctions
 - LISP, Scheme, CaML, Haskell
- Objet : données et actions sur les données sont liées
 - Java, C#, Smalltalk
- Certains langages sont multi-paradigmes (Python...)

Contraintes sur les manipulations de données

- Déclaratif : il faut déclarer les types des données (entier, texte, tableau de nombre entiers, ...)
- Fortement typé : il faut convertir explicitement les types de données
- Python est fortement typé mais non déclaratif

Une classification des langages de programmation

Langages « près de la machine »

- Ils imposent de gérer les allocations de mémoire « à la main »
- Toujours compilés
- Conceptuellement proches de l'assembleur, mais bien plus commodes
- Ils sont très performants mais nécessitent une forte expertise
- Nécessaires pour écrire systèmes d'exploitation, certaines bibliothèques et utilitaires systèmes
- Adaptés au calcul intensif, au temps réels, à certains jeux
- Exemples : *C*, *C++*, *D*, *Pascal*, *PL/1*, *Ada*

Les implémentations (compilateurs, interpréteurs) des autres types de langages sont généralement écrits dans un des langages ci-dessus.

Une classification des langages de programmation

Langages de « haut niveau » mathématiques

- Gèrent les allocations mémoire automatiquement
- Généralement interprétés (tournent sur une *machine virtuelle*)
- Reposent sur des concepts mathématiques avancées
- Nécessitent une forte expertise et un esprit mathématique
- Exemple : *LISP, Scheme, CaML, Haskell*

Langages de « haut niveau » pragmatiques

- Comme ci-dessus mais plus facile d'accès
- Parfois sacrifient la maintenabilité (facile d'écrire du code pas très propre) : *BASIC, PHP, JavaScript*
- D'autres incitent aisément à écrire du code lisible et maintenable : *Ruby, Python*

Nous travaillerons avec Python 3 dans ce cours.

Histoire de Python

- Créé en 1991 par Guido Van Rossum
- Vise à être lisible, rigoureux, souple et expressif
- Convient autant aux débutants qu'aux développeurs confirmés
- Python 2 s'est popularisé dans les années 200x
- Python 3 existe depuis une décennie
- Logiciel libre (*Open Source*)

Caractéristiques de Python

- Orienté objet
- Multi-paradigme : impératif, fonctionnel, objet
- Portable : UNIX, Linux, MS Windows, ...
- Propose un grand nombre de bibliothèques d'extension (modules)

Lexique et grammaire

Pour programmer il faut être très précis

Lexique

Un compilateur ou un interpréteur c'est « bête et méchant »

- Il reconnaît certains mots : *if*, *else*, *def*
- Il interprète d'autres mots spécialement : *print*, *help*
- Il détecte les mots que vous définissez : *price = 42*

Grammaire

Il reconnaît certaines constructions grammaticales, par exemple :
« *if* suivi d'une expression, puis *then*, puis : (deux points), puis retour à la ligne et espaces en début de lignes »

```
if price > 42:  
    print("C'est cher !")
```

Instructions et expressions

Python distingue les instructions des expressions

Une expression est un calcul et *a une valeur* :

```
42 + 13  
"Bonjour " + name + " !"  
len(foodInfo)
```

Valeurs : 55, 'Bonjour John !' (si *name* correspond à 'John') et la longueur d'une structure de données.

Une expression peut avoir un effet secondaire

i.e. « faire quelque chose en plus d'avoir une valeur »

```
print('Bonjour !')
```

ici la valeur est *None*, une valeur un peu spéciale en Python signifiant « rien ».

Instructions et expressions

Une instruction *fait* quelque chose

Une instruction n'a qu'un « effet secondaire » elle n'a pas de valeur

```
res = res * i      # instruction d'affectation
if price < 42:     # ces deux lignes forment une
    print("C'est pas cher !")    # instruction
```

Une instruction peut contenir des expressions

```
msg = 'Bonjour ' + name
if price > 12 and price < 56:
    print("C'est ok.")
```

- 'Bonjour ' + name est une expression
- price > 12 and price < 56 aussi
- print(...) aussi

L'inverse est impossible.

Erreurs courantes

Signalée par le compilateur

Utiliser une instruction là où une expression est attendue : l'erreur sera signalée

```
if i = 1:
    ...
SyntaxError: invalid syntax
```

Non signalée par le compilateur (au mieux par un *warning*)

Utiliser une expression comme si c'était une instruction : le calcul aura lieu mais n'aura aucun effet pratique : c'est un BUG !

```
31 + 11
if price == 42:
    print('Bon prix !')
```

Correction :

```
price = 31 + 11
```

Langage compilés

Le compilateur analyse le code, le traduit en assembleur, puis en binaire

```
$ gcc -o factorielle factorielle.c
$ file factorielle
factorielle: ELF 64-bit LSB pie executable,
x86_64, version 1 (SYSV), dynamically linked,
...
$ ./factorielle
Nombre : 5
120
```

On peut examiner le code assembleur intermédiaire

```
$ gcc -S factorielle.c
$ less factorielle.s
```

Langages interprétés

Bytecode

- La plupart des langages interprétés sont compilés vers un format particulier : le *bytecode*
 - C'est un format binaire
 - Non lié à une architecture matérielle de CPU
 - Conçu pour être exécuté par un logiciel : une *machine virtuelle*
- Exemples : Java, C#, Python, Perl, ...

Exemples de machines virtuelles

- JVM : *Java Virtual Machine* (Java, Groovy, ...)
- .NET *Runtime* (C#, VisualBasic.NET, F#, ...)
- PVM : *Python Virtual Machine*
- Perl 6 : *Parrot*, *MoarVM*

Compilation et exécution

Les deux étapes peuvent être séparées

```
$ javac hello.java  
$ ls  
hello.java hello.class  
$ java hello  
Bonjour le monde !
```

... ou bien enchaînées automatiquement

```
$ python hello.py  
Bonjour le monde !
```

Bytecode

Désassemblage du bytecode

En Python on peut examiner le code pseudo-assembleur intermédiaire

```
...
>> 20 FOR_ITER                12 (to 34)
      22 STORE_FAST            2 (i)

4      24 LOAD_FAST            1 (res)
      26 LOAD_FAST            2 (i)
      28 BINARY_MULTIPLY
      30 STORE_FAST            1 (res)
      32 JUMP_ABSOLUTE        20
>> 34 POP_BLOCK

....
```

Enrichissent le langage

- Vous n'êtes pas le premier à vouloir calculer une factorielle
- Des bibliothèques permettent d'étendre les fonctionnalités d'un langage
 - Calculs mathématiques
 - Manipulation de texte
 - Interfaces graphiques, programmation pour le Web
 - Bases de données
 - etc.
- Les systèmes d'exploitation comportent de nombreuses bibliothèques binaires (.dll sous MS Windows, .so sous UNIX et Linux)
- D'autres peuvent être installées par la suite

Bibliothèques pour Python

Modules Python

- Python est fourni « avec les piles » : nombreux modules standards
 - Mathématiques, manipulation de texte, réseau, système, etc.
 - Une section entière du manuel leur est consacrée
- Des modules supplémentaires sont publiés sur <http://pypi.org>

Exemple : le module *math*

```
$ python
>>> factorial(5)
...
NameError: name 'factorial' is not defined
>>> from math import factorial
>>> factorial(5)
120
```

Programmer avec Python

Le mode interactif

- Il suffit d'installer Python 3 pour pouvoir commencer !
 - Sous MS Windows ne pas oublier l'option de modification du PATH

- Ouvrir un terminal

```
$ python
```

```
>>> 42 + 13
```

```
55
```

```
>>> price = 42
```

```
>>> price = price / 2
```

```
>>> price > 16 and price < 42
```

```
True
```

```
>>> print('Bonjour tout le monde !')
```

```
Bonjour tout le monde !
```

Un premier programme

Créez un fichier `hello.py` avec un éditeur de texte

```
name = input('Votre nom : ')\nprint('Bonjour ' + name)
```

Dans un terminal, en se plaçant dans le répertoire du fichier

```
$ cd ~/devel/Python\n$ pwd\n/home/john/devel/Python\n$ ls\nhello.py\n$ python hello.py\nVotre nom : John\nBonjour John
```

Le mode Python interactif

Le mode interactif

- Utilisable en exécutant `python` (ou `python3`) sans arguments
- Il en existe des variantes : *idle*, *bpython*, *jupyter*, ...
- On l'appelle la boucle REPL :
 - ① **R**ead : lire la saisie de l'utilisateur
 - ② **E**val / **E**xecute : évaluer l'expression / exécuter l'instruction
 - ③ **P**rint : afficher la valeur si c'est une expression
 - ④ **L**oop : on retourne à l'étape 1.
- Une exception à l'affichage (*Print*) : quand la valeur de l'expression est *None*

```
>>> print('Bonjour')
Bonjour
>>> print(print('Bonjour'))
Bonjour
None
```

Exécution d'un programme Python

En ligne de commande sous Windows ou UNIX/Linux

Il suffit d'indiquer à python le fichier à exécuter

```
$ python hello.py
```

```
Bonjour tout le monde !
```

En ligne de commande sous UNIX/Linux

- Si le fichier débute par une ligne spéciale

```
#!/usr/bin/env python3
```

```
print('Bonjour tout le monde !')
```

- On peut alors donner le droit d'exécution sur le fichier

```
$ chmod +x hello.py
```

```
$ ls -l hello.py
```

```
-rwxr-xr-x 1 john  john ...    hello.py
```

```
$ ./hello.py
```

```
Bonjour tout le monde !
```

Exercice : *Ok boomer!*

Écrire un premier programme simple

- Demande son nom et son âge à l'utilisateur
- Modifiez le fichier `hello.py` ou copiez le en `boomer.py`

```
name = input('Votre nom :')  
age  = int(input('Votre âge :'))
```
- Salue l'utilisateur et affiche « *Boomer!* » si l'âge est supérieur à votre âge

```
$ ./hello.py  
Votre nom : John  
Votre âge : 42  
How are you John?  
Boomer!
```

Bonnes pratiques de programmation

En général

- Commentez votre code : en Python tout ce qui suit un dièse «#» est ignoré par le compilateur
- Nommez vos variables en rapport avec leur *sens* : *name*, *price*, *products*, et non pas *a*, *b*, *data*
- Lisez les messages d'erreur à la compilation et à l'exécution

Plus particulièrement en Python

- Documentez vos scripts, et plus tard vos fonctions, vos classes, vos modules
- Il suffit d'insérer du texte en début de bloc, encadré par trois apostrophes ou trois guillemets droits

```
#!/usr/bin/env python3
'''Ce script demande son âge à l'utilisateur
et le salue de façon ironique.'''
```
- C'est plus qu'un commentaire, un `help()` reprend ce texte.

Spécificités de Python

Définition de bloc de code par l'indentation

- Vous avez remarqué que la définition d'un bloc (après un *if* par exemple) se fait en Python en insérant *deux points* et en passant à la ligne, puis en *indentant* la suite
- Il est fortement recommandé d'utiliser 4 espaces
- La fin du bloc est marqué par un retour au niveau précédent
- C'est différent dans d'autres langages
 - { et } en C, C++, Java, C#, Perl, PHP, ...
 - (et) en LISP et Scheme, Begin et End en Pascal
 - et bien d'autres possibilités encore...

```
if age > 42:
    print('Boomer!')
    print('No offense.')
print('Bye ' + name + '!')
```

Spécificités de Python

Pas de déclaration des variables

- Il suffit de les affecter : `name = 'John'`, pour qu'elles existent
- Cependant Python est fortement typé : vérification des compatibilités de types, pas de conversion automatique (en général)

```
>>> '42' > 32
```

```
TypeError: '>' not supported between instances  
of 'str' and 'int'
```

```
>>> int('42') > 32
```

```
True
```

Évaluer un nom inexistant ?

```
>>> product
```

```
NameError: name 'product' is not defined
```

Données et objets

Données

- *Objet* stocké en mémoire par la *Python Virtual Machine*
- La mémoire est allouée quand nécessaire
- La mémoire pourra être libérée plus tard si la donnée n'est plus utilisable (référéncée)
- Un objet a toujours un *type*

```
>>> 42
42
>>> type(42)
<class 'int'>
>>> type('John')
<class 'str'>
>>> id(42)          # adresse en mémoire
9080320
```

Objets et noms

On peut créer un nom en lui affectant un objet

```
>>> name = 'John'
>>> age = 42 + 13
>>> type(name)
<class 'str'>
>>> type(age)
<class 'int'>
>>> id(name)
140240346121136
```

Un nom peut être supprimé

```
>>> del(age)
>>> age
NameError: name 'age' is not defined
```

Références

Un nom est toujours en Python une *référence*

- Au sens strict ce n'est pas une *variable*
- Plusieurs noms peuvent référencer le *même* objet

```
>>> food = 'spam'
>>> bad = food
>>> id(food)
140240345726680
>>> id(bad)
140240345726680
>>> food is bad
True
```

Comparer deux références

- `is` pour tester si c'est *le même objet* en mémoire
- `==` pour tester si deux objets référencés ont la même *valeur*

Types numériques

Famille de types comparables

- `int` : nombres entiers, signés, de taille quelconque
- `float` : nombres à virgules, limites sur la taille et la précision
- `complex` : nombres complexes
- On peut les combiner dans les opérations mathématiques usuelles, les comparer

```
>>> price = 42
```

```
>>> maximum = 49.99
```

```
>>> length = 42.0
```

```
>>> z = 1 + 2j
```

```
>>> price <= maximum           # inférieur ou égal
```

```
True
```

```
>>> price == length
```

```
True
```

```
>>> (z - 1) ** 2 == -4
```

```
True
```

Opérations numériques

Arithmétique

- $+$, $-$, $*$ (multiplication), $/$, $//$ (division entière), $**$ (exponentiation)
- $\%$: *modulo* (reste de la division euclidienne)
- $\sim$, $\wedge$, $|$, $\&$: opérations bit-à-bit
- Correspondent à des méthodes « doublignées » (*dunder methods*) : `__add__`, `__mul__`, ...
- Les nombres complexes ont des attributs : `z.real`, `z.imag`

Les priorités habituelles s'appliquent, on peut regrouper des sous-expressions entre parenthèses.

Type chaîne de caractère

Chaînes de caractères

- `str` : collection de caractères
- Écrites littéralement encadrée par des guillemets droits, des apostrophes ou trois consécutifs de l'un ou l'autre

```
>>> name = '''John'''
>>> msg  = "Ce n'est pas Joe"
>>> text = 'Il est un "boomer"'
```
- Comme dans beaucoup d'autres langages le *backslash* a un sens particulier
 - Pour insérer un caractère d'encadrement `'c\'est un "boomer"'`
 - Pour insérer un caractère spécial : retour chariot `\n`, smiley `\N{grinning face with smiling eyes}`, tabulation `\t`
- Notez qu'en Python le *backslash* en dehors des chaînes permet de passer à la ligne sans que Python ne le prenne en compte
- `chr(n)` retourne le caractère de code `n`, `ord(c)` le code du caractère `c`

Opérations sur les chaînes

Opérateurs

- `+` : concaténation
- `*` : répétition (multiplication par un entier)
- `in` : recherche si une sous-chaîne est présente

```
>>> 'John' + ' ' + 'Cleese'
'John Cleese'
>>> 'spam ' * 4
'spam spam spam spam '
>>> 'spam' in 'ham egg spam sausage'
True
```

Fonctions

- `len()` : longueur, comme pour toutes les collections
- `int()` : essaye de convertir en entier (erreur si impossible)
- `float()` : essaye de convertir en flottant (erreur si impossible)

Opérateurs et affectations

Il est courant d'utiliser un opérateur et de réaffecter le résultat au nom de départ

```
>>> food = 'egg '  
>>> food = food + ' spam'  
>>> food  
'egg spam'  
>>> age = 42  
>>> age = age + 1
```

Un raccourci : *opérateur=*

```
>>> food += ' spam'  
>>> age += 1  
>>> qty *= 2  
>>> price /= 2
```

Une *fonction* est un objet qu'on peut *appeler*

```
>>> len      # objet
<built-in function len>
>>> type(len)  # il a un type
<class 'builtin_function_or_method'>
>>> len('spam') # appel : func()
4
```

Une *méthode* est une fonction dans l'espace d'un objet

Peut *modifier* l'objet concerné ou *retourner* un autre objet.

```
>>> food = 'spam'
>>> food.upper()      # nouvel objet
'SPAM'
>>> food.isupper()
False
>>> food              # inchangé
'spam'
```

Construire une chaîne complexe

Combiner éléments constants et variables

```
>>> price = 42; prod = 'spam'
>>> "Le prix du " + prod + ' est ' \
    + str(price - 10) + '€'
'Le prix du spam est 32€'
```

Plus commode : la méthode str.format()

```
>>> 'Le prix du {} est {}€'.format(prod, price - 10)
```

Depuis Python 3.7 : encore plus simple !

```
>>> f"Le prix de {prod} est {price - 10}€"
```

Nombreux paramétrages possibles, voir le manuel sur *str.format* et sur les « *f-strings* »

Comment retrouver toutes les fonctions et méthodes ?

Il ne faut pas hésiter à consulter le manuel !

- <http://docs.python.org/3.7/>
- Le manuel simplifié est accessible via `help()`

On peut demander à Python de nous les montrer !

```
>>> dir(__builtins__) # Noms intégrés de base
[ ...'dir', ... , 'int', ..., 'len', ..., 'str', ... ]
>>> dir(str) # méthodes sur les chaînes
[ ..., 'lower', ..., 'isupper', ..., 'upper', ... ]
>>> import math
>>> dir(math)
[ ..., cos , factorial, ..., pi, ..., sin, ... ]
>>> from math import sin,cos,pi
>>> sin(pi/4) + cos(pi/4)
1.414213562373095
```

Booléens

Type bool

- Deux valeurs possibles True et False
- Renvoyés par les opérateurs de comparaison ==, !=, <, <=, >, >=, is, is not
- On peut combiner des comparaisons avec or, and, not pour construire des expressions logiques

Que fait `if expression:` ?

- Il convertit l'expression en booléen : `bool(expression)`
- Pour 0, 0.0, '', None `bool(...)` est *False* sinon *True*
- De même pour tout ce qui est « vide » (collections)
- Si vrai (*True*), if exécute le bloc de code qui suit

```
price = 42
if price:
    print("C'est pas gratuit...")
```

La condition if

if peut être suivi d'un bloc else

```
if 'spam' in food:
    food += ' more spam'
else:
    food += ' spam' # spam obligatoire !
```

Et aussi d'une suite de elif (*else if*)

```
if 'spam' in food:
    food += ' more spam' # more spam!
elif 'egg' in food:
    food += ' spam'      # free spam!
else:
    food += ' spam spam' # double spam!
```

if en tant qu'expression

La forme `if: ...: ... else:` est une *instruction*

Il est possible d'exprimer une alternative comme une expression

```
>>> price = 42
>>> 'cher' if price > 50 else 'bon marché'
'bon marché'
>>> price = 57
>>> 'cher' if price > 50 else 'bon marché'
'cher'
```

Simple, concis et lisible : expressif !

```
msg = 'Votre panier contient {} produit{}'.format(n,
          's' if n > 1 else '')
```

Exercice : *English Breakfast*

Passer notre commande

- Si du *spam* est commandé ajouter *good!*, sinon si *ham* est commandé ajouter *spam* et enfin, sinon (ni *spam*, ni *ham* ne sont commandés) ajouter *spam spam*
- Notez que ce qui suit «Commande : » est une saisie de l'utilisateur, le reste est affiché par le programme.

```
$ python3 breakfast.py
```

```
Commande : ham egg
```

```
Assiette : ham egg spam
```

```
$ python3 breakfast.py
```

```
Commande : ham sausage
```

```
Assiette : ham sausage spam
```

```
$ python3 breakfast.py
```

```
Commande : egg spam
```

```
Assiette : egg spam good!
```

Modifier le programme pour ajouter, en *plus*, 10 fois du *spam* si jamais *coffee* est demandé

```
>>> ' covfefe' * 3  
    ' covfefe covfefe covfefe'
```

Compter et afficher la quantité de *spam* servie

La méthode count sur les chaînes permet de compter les occurrences d'une sous-chaîne

```
>>> 'to be or not to be'.count('to')  
2
```

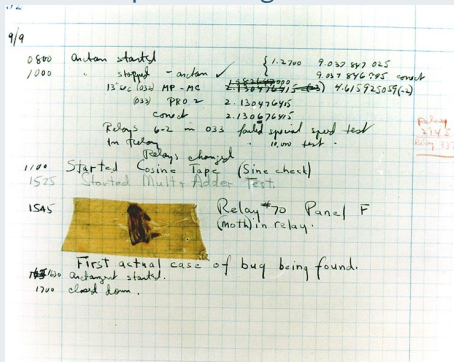
Modifier le programme pour afficher la quantité de *spam* servie

Ajoutez le code à la suite de du bloc if : elif: else:

```
$ python breakfast.py  
Commande : egg sausage  
Assiette : egg sausage spam spam  
Spam      : 2
```

Et si on commande un hamburger ?

- Testez en commandant un hamburger
- Combien de spam recevez vous ? Pourquoi ?
- Et voilà, c'est notre premier *bug*...



>>> 'ham' in 'egg hamburger coffee'

True

- cf. [https://fr.wikipedia.org/wiki/Bug_\(informatique\)](https://fr.wikipedia.org/wiki/Bug_(informatique))

Collections

Objets contenant des références à d'autres objets

```
>>> menu
[ 'ham', 'spam', 'egg', 'sausage' ]
>>> menu[0] + menu[2]
'hamegg'
>>> prices
{ 'ham': 42, 'spam': 12, 'sausage': 20 }
>>> prices['spam']
12
```

Parmi elles, un premier cas : les *listes*

```
>>> menu = [ 'ham', 'spam', 'egg', 'sausage' ]
>>> menu[0]
'ham'
>>> len(menu)
4
```

Listes

On peut les créer entre crochets [*expression*, ...]

- Expressions séparées par des virgules
- Peuvent contenir n'importe quels types : *str*, *int*, *list*, etc.
- On peut en obtenir facilement à partir d'une chaîne

```
>>> data = [ 'a', 42, 12, -3.14, cos(pi) ]
>>> table = [ [ -2, 4 ], [ 7, 0 ] ]
>>> food = 'ham spam egg bacon spam'.split()
>>> help(str.split)
```

On peut accéder aux *éléments* avec un index entier

Le *premier* élément est à l'indice 0

```
>>> data[0]
'a'
>>> table[1][0]
7
```

Extraction d'éléments et modification d'une liste

On peut modifier les références stockées dans une liste

```
>>> food[3] = 'sausage'
```

Les indices négatifs partent de la *fin*

```
>>> food[-1]
```

```
'spam'
```

```
>>> food[-2]
```

```
'sausage'
```

On peut extraire une sous-liste (« tranches » – *slices*)

```
>>> food
```

```
['ham', 'spam', 'egg', 'sausage', 'spam']
```

```
>>> food[2:4]
```

```
[ 'egg', 'sausage' ]
```

Notez que l'on a extrait une tranche allant de l'élément d'indice 2 jusqu'à l'élément d'indice $4 - 1 = 3$ (donc 4 non inclus)

Opérations sur les listes

Les opérateur +, * et in/not in

```
>>> food = [ 'ham', 'spam' ]
>>> food + [ 'egg', 'sausage' ]
[ 'ham', 'spam', 'egg', 'sausage' ]
>>> food * 2
[ 'ham', 'spam', 'ham', 'spam' ]
>>> 'spam' not in food
False
>>> 'cheese' in food
False
```

On peut construire une chaîne à partir d'une liste de chaînes

```
>>> ' ; '.join(food * 2)
'ham ; spam ; ham ; spam'
```

Notez (avec surprise ?) que ce n'est pas list.join(sep) mais sep.join(list)

Exercices : des chaînes aux listes

Reprenons le résultat de l'exercice précédent

- Décomposez la commande passée par saisie de l'utilisateur dans une liste
- Faites porter tous les tests qui suivent sur cette liste
- Faut-il changer du code ? Où ? Comment ?
- La méthode *count* existe-t-elle sur les listes ? Fonctionne-t-elle comme espérée ?
- Affichez la commande passée à la fin du script, sous cette forme :

```
**** Fawltz Towers Hotel ****
ham
...
bacon
**** Service not included ****
```

- Le bug du *hamburger* est-il toujours là ? Pourquoi ?

Exercice : Introspection sur les listes

Examinez les méthodes sur les listes

Ignorez, pour l'instant, celles dont les noms sont encadrés par __

```
>>> myFood = [ 'spam', 'egg' ]
>>> dir(list)
...
>>> help(list.index)
>>> myFood.index('spam')
```

Déterminez celles qui modifient la liste concernée quand on les appelle et celles qui renvoient simplement un autre objet

Que fait la méthode `sort` sur les listes ? Comparez avec la fonction `sorted`.

Modification d'une liste

On peut changer un élément d'une liste

```
>>> food = [ 'spam', 'ham', 'egg' ]  
>>> food[1] = 'sausage'  
>>> food  
[ 'spam', 'sausage', 'egg' ]
```

Des méthodes permettent aussi de les modifier

```
>>> food.append('pudding')  
>>> food.remove('spam')  
>>> food.pop()
```

insert, reverse, sort, extend, clear

Affectations sur les tranches

Une tranche peut se placer à *gauche* d'une affectation

```
>>> food = 'spam ham egg sausage cheese'.split()
>>> food[1:3]
['ham', 'egg']
>>> food[1:3] = []
>>> food
['spam', 'sausage', 'cheese']
>>> food[1:2]
['sausage']
>>> food[1:2] = [ 'spam', 'pudding', 'beans' ]
>>> food
['spam', 'spam', 'pudding', 'beans', 'cheese']
```

Déballage de séquences

Extraire des données d'une séquence dans des noms

```
>>> product = [ 'spam', 42 ]
>>> food, price = product
>>> food
'spam'
>>> price
42
```

On peut profiter des tranches ou récupérer les restes

```
>>> product = [ 'spam', 42, 'good', 10 ]
>>> food, price = product[:2]
>>> food, price, *end = product
>>> end
['good', 10]
```

Autres collections Python : dictionnaires

Dictionnaires : clés et valeurs

```
>>> pricedb = { 'spam':12, 'ham':42, 'egg':10 }
>>> pricedb['ham']
42
>>> 'ham' in pricedb
True
```

Modifiables

```
>>> pricedb['beans']
KeyError: 'beans'
>>> pricedb['beans'] = 7
>>> pricedb['beans']
7
>>> pricedb.get('spam')
12
>>> pricedb.get('tomatoes',0)
0
```

Construction de dictionnaires

Accolades et fonction dict()

```
>>> { 'first':'John', 'last':'Doe', 'age':42 }  
>>> dict( [ ['first', 'John'], ['last','Doe'], \  
            ['age',42] ] )  
>>> dict(first='John', last='Doe', age=42)
```

zip() apparie les termes d'une séquence

```
>>> l1 = [ 'first', 'last', 'age' ]  
>>> l2 = [ 'John', 'Doe', 'age' ]  
>>> d = dict(zip(l1,l2))
```

update() fusionne avec un autre dictionnaire

```
>>> d.update( { 'food':'spam', age:'43' } )
```

Autres collections Python : tuples

Tuple : séquence non modifiable (immuable)

On accède aux éléments exactement comme pour des listes (index, tranches, déballage)

```
>>> foods = ( 'spam', 'ham', 'egg', 'beans' )
```

```
>>> prices = ( 12, 42, 10, 7 )
```

```
>>> foods[2]
```

```
'egg'
```

```
>>> foods[1:3]
```

```
( 'ham', 'egg' )
```

```
>>> prices[1] = 44
```

```
TypeError: 'tuple' object does not support item  
assignment
```

```
>>> prices_list = list(prices)
```

```
>>> prices_list[1] = 44; prices_list  
[ 12, 44, 10, 7 ]
```

Autres collections Python : ensembles

set et *frozenset* : ensembles modifiables et non modifiables

```
>>> food = { 'spam', 'ham', 'egg', 'ham' }
>>> len(food)
3
>>> food
{ 'spam', 'ham', 'egg' }
```

Supprimer des doublons dans une liste

```
>>> food = [ 'spam', 'ham', 'egg', 'ham', 'spam' ]
>>> food = list(set(food))
>>> food
['ham', 'spam', 'egg']
```

Immuables et modifiables

Parmi les types que nous avons manipulés lesquels sont immuables et lesquels sont modifiables ?

Immuables

- Nombres : *int*, *float*, *complex*
- Chaînes : *str*
- Tuples : *tuple*
- Ensembles gelés : *frozenset*
- Booléens *bool*, *NoneType*

Modifiables

- Listes : *list*
- Dictionnaires : *dict*
 - Mais leurs clés *doivent* être d'un type immuable
- Ensembles : *set*

Conversions de types

On peut convertir facilement d'un type à un autre

```
>>> int('42')
42
>>> list('spam')
['s', 'p', 'a', 'm']
>>> tuple([ 1, 2, 3 ])
(1, 2, 3)
>>> list( (1,2,3) )
[1, 2, 3]
>>> ''.join([ 's', 'p', 'a', 'm'])
'spam'
```

Tous les types (*str*, *int*, *list*, ...) sont des fonctions

- Convertissent au mieux (et parfois échouent avec une erreur) vers le type concerné
- Sans argument renvoient le vide, le rien, le zéro, ...

Boucle for

L'instruction for parcourt une collection

```
>>> foods = 'spam egg ham'.split()
>>> for food in foods:
    print(food)

spam
egg
ham
```

Fonctionne pour tous les collections (et *itérables*)

- Listes et leurs tranches
- Tuples et leurs tranches
- `range(n,m,p)` : entiers de n à $(m-1)$ avec un pas de p
- Dictionnaires (parcourt les *clés*)
- Ensembles
- et aussi (*itérables*) : fichiers, requêtes à des bases de données, lecteurs de fichiers CSV, etc.

Boucles et déballage

Les éléments peuvent être des séquences (listes, tuples)

```
>>> foods = [ ('spam', 12), ('ham', 42) ]
>>> for elt in foods:
    print(elt)                # une séquence (tuple)
    food, price = elt         # éléments
    print(food,price)
('spam', 12)
spam 12
('ham', 42)
ham 42
```

for peut parcourir et déballer

On peut parcourir et déballer en même temps

```
>>> foods = [ ('spam', 12), ('ham', 42) ]
>>> for food, price in foods:
        print(food,price)
spam 12
ham 42
```

Parfait pour les dictionnaires !

```
>>> prices = { 'spam':12, 'ham':42 }
>>> for food, price in prices.items():
        print(food,price)
spam 12
ham 42
```

Pour un fichier texte `chiffres.txt` de la forme :

un	1
deux	2
...	
neuf	9

Il suffit d'une boucle *for*

with s'assure que le bloc ne soit pas exécuté si l'ouverture échoue, et sinon ferme le fichier proprement.

```
with open('chiffres.txt') as input:
    for line in input:
        word, value = line.rstrip().split()
        value = int(value)
        print('{} se dit {} ' \
              'en français'.format(value, word))
```

C'est encore plus simple pour un fichier CSV ou une base de données (modules `csv` et modules DB API pour les bases SQL)

Changeons de méthode

Qu'est-ce que le *peer programming* ?

- Programmation en binôme
- En groupe de deux, chacun tour à tour *conducteur* (écrit du code) ou *observateur* (assistant demandant des explications, suggérant des pistes de conception, ...)
- **Seul** le conducteur peut toucher son propre clavier
- D'un exercice à l'autre inversez les rôles
- Il y a quatre exercices, prenez le temps d'en faire deux, ou trois, ou quatre

Exercice : *Menu à la carte*

Dans le programme `alacarte.py` s'inspirant du précédent

Écrire un programme qui demande à nouveau une suite d'aliments et les stocke dans une liste.

- Affichez les aliments un par un dans une boucle `for`
- Affectez une liste vide à `up_foods`
- Dans une boucle `for` qui parcourt la liste de commande ajoutez à `up_foods` chaque aliment modifié en majuscules
- Affichez les éléments de la liste `up_foods`

La facture...

Créez un fichier `prices.txt` de la forme :

```
ham      42
spam     12
beans    8
sausage  11
cheese   10
...
```

La facture (suite)

La facture !

- Lisez ce fichier et stockez le dans un dictionnaire (indice : un dictionnaire vide est {})

```
with open(...) as ...:
    pricedb = {}
    for line in ...:
        prod, price = line.rstrip().split()
        price = float(price)
        pricedb[prod] = price
    print(price)
```

- Affichez la facture avec les prix de chaque ingrédient et affichez aussi le prix total du petit déjeuner

Exercice : Crible d'Érathostène

Initialisez l'algorithme d'Érathostène

- Demander un nombre n à l'utilisateur
- Construire une liste d'entiers de 2 à l'entier n inclu
`numbers = list(range(2,n+1))`
- Partez de $k = 2$

Utilisez une boucle pour la suite

- Barrer une case étant exprimé par une mise à zéro ou une suppression avec la méthode de liste `remove()`
- Considérez la condition de sortie de boucle

Résultat du Crible d'Érathostène

Affichez le résultat

- La suite des nombres non nuls ou encore présents
- Utilisez une boucle pour cela

Exercice : Cryptographie

Construisez l'alphabet

- Dans une boucle faites parcourir à i :
`range(ord('a'), ord('a') + 26)`
- Affichez d'abord `chr(i)`
- Modifiez la boucle pour ajouter `chr(i)` dans une liste :
`alphabet = []`
`for i in range(ord('a'), ord('a') + 26):`
 # ajoute chr(i) à la liste alphabet
 ... *# à vous : alphabet.append(...) ou +=*
`print(alphabet)`

Jouons avec le code d'une lettre

Soit une lettre dont le code ASCII est k - Que calcule ceci ?

- ```
chr((k - ord('a') + 5) % 26 + ord('a'))
```
- Essayez avec  $k$  valant `ord('e')` et `ord('x')`

# Exercice : Cryptographie (suite)

## Chiffrons !

- Demandez une phrase à l'utilisateur en début du même script
- Parcourez la chaîne reçue, pour chaque caractère vous afficherez l'élément de alphabet d'indice :

```
chr((ord(c) - ord('a') + 5) % 26 + ord('a'))
```

- Utilisez pour l'affichage : `print(..., end = '')`

```
for c in text.lower():
 if c in alphabet:
 print(..., end = '')
 else:
 print(c)
print() # pour passer à la ligne
```

- Il y avait plus simple pour obtenir alphabet :

```
>>> import string
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
```

# L'autre boucle : while

## Une autre instruction de boucle : while

Exécute un bloc tant qu'une condition est vraie

```
>>> food, menu = '', []
>>> while food != 'spam':
... food = input('Aliment : ')
... menu.append(food)
... print('Demander du spam pour finir...')
...
Aliment : ham
Aliment : egg
Aliment : spam
>>> print(menu)
['ham', 'egg', 'spam']
```

# Contrôle de flot d'exécution

## On peut sortir d'une boucle prématurément : *break*

- Quand on a rencontré une anomalie
- Quand on a trouvé ce que l'on cherchait
- Quand l'utilisateur indique qu'il veut quitter

```
while True:
 ans = input('Nom (!END pour quitter) : ')
 if ans == '!END':
 break
```

## On peut sauter la suite du bloc avec *continue*

- Quand on souhaite ignorer une valeur

```
for line in text:
 if line.startswith('#'):
 continue
 data = line.strip().split()
 # Traitement des données
```

# Compréhensions

## Une construction différente avec for

Construit une liste à partir d'une autre de façon descriptive

```
>>> foods = ['spam', 'ham', 'egg', 'beans']
>>> [food.upper() for food in foods]
['SPAM', 'HAM', 'EGG', 'BEANS']
```

## On peut filtrer en amont

```
>>> [food.upper() for food in foods \
 if food != 'egg']
```

## Et aussi introduire une alternative

Le *if ... expression else ...* déjà aperçu.

```
>>> [food.upper() if food != 'spam' else 'beuh' \
 for food in foods \
 if food != 'egg']
```

# Fonctions

Une fonction permet de nommer et réutiliser un bloc de code

Définie par l'instruction `def`

```
>>> def printLicense():
 print('Distribuable selon les termes')
 print('de la licence WTFPL')
 print('voir : http://www.wtfpl.net/')

>>> printLicense()
Distribuable selon les termes
de ...
```

Il suffit de la définir dans un fichier `wtfpl.py` pour avoir un module utilisable

```
>>> from wtfpl import printLicense
>>> printLicense()
```

# Arguments et valeur de retour

## Une fonction peut attendre des arguments

```
>>> def cry_if_spam(string):
 if 'spam' in string:
 print("I do not like SPAM!!!")
>>> cry_if_spam('ham egg')
>>> cry_if_spam('ham spam egg')
I do not like SPAM!!!
```

## Une fonction peut renvoyer quelque chose (autre que *None*)

```
>>> def where_is_spam(menu):
 return menu.index('spam')
>>> where_is_spam(['egg', 'ham', 'spam', 'beans'])
2
```

# Arguments optionnels

Il suffit de spécifier une valeur par défaut

```
>>> def price_of(food, discount = 0):
 prices = { 'spam': 12, 'ham': 42 }
 price = prices.get(food, 0)
 price *= (100 - discount)/100
 return price
>>> price('ham',10)
37.8
>>> price('ham')
42
```

# Appels de fonctions

On peut passer les arguments par position

```
>>> price_of('food', 10)
```

On peut aussi passer les arguments par mots-clés

```
>>> price_of(discount = 20, food = 'spam')
```

On peut combiner les deux, *cf.* la fonction *print*

```
>>> print('spam', 'ham', 'egg', 42, sep='\n')
```

# Arguments en nombre variables

On peut *collecter* les arguments passés par position

```
def all_up(*args):
 return [elt.upper() for elt in args]
```

De même pour les arguments passés par mot-clés

```
def show_data(**kwargs):
 # kwargs est un dictionnaire
 for key, val in kwargs.items():
 print('{} -> {}'.format(key, val))
```

# Déballage de collections dans des arguments

À l'inverse on peut *déballer* une séquence dans une suite d'arguments

Il seront passés à la fonction par *position*

```
print(*['spam', 'ham', 'egg', 'bacon'])
```

De même pour un dictionnaire, déballé dans des arguments passé par mots-clés

```
>>> d = { 'name': 'John', 'age': 42 }
>>> show_data(*d)
```

# Portée des noms

## Quand vous dérèférez un nom

- Python regarde d'abord dans la portée *locale*
- Puis dans une éventuelle fonction *englobante*
- Enfin *globalement* (script ou module)
- Et en dernier lieu les noms *built-in* (intégrés)
- Règle **LEGB**

## Quand vous affectez un nom

- Instructions = ou *opérateur*= (ce dernier pas toujours...)
- Ça ne concerne que la portée locale, c'est un nouveau nom
- On peut outrepasser cette règle, c'est une *mauvaise* idée

## Paramètres modifiables

- Une fonction reçoit ses argument par *références*
- Si un argument est modifiable et modifié (ex : *data.append(...)*) la modification se répercute à l'extérieur

## Exercice : *Professional Breakfast*

### En partant de la solution de l'exercice précédent

La solution de l'exercice précédent lit les informations à partir d'un fichier. Solicitez votre formateur s'il vous manque des éléments (fichier, solution).

- ❶ Écrivez une fonction qui prend un aliment en argument et retourne son prix
- ❷ Écrivez une fonction qui prend une liste d'ingrédients en argument et retourne le prix total
- ❸ *print* affiche un objet liste d'une façon sommaire (et utile pour la mise au point), regardez ce qui se passe si vous préfixez la liste par le caractère \*

### Bonus (si vous avez le temps)

Ajoutez le caractère \* devant le paramètre de la fonction définie à l'étape 2, appelez maintenant la fonction avec les aliments passés en paramètres, quels que soient leur nombre.

# Mise au point et débogage

## Interpréter correctement les messages d'erreur

- Python indique généralement l'erreur au plus près du *vrai* problème
- Interpréter correctement les exceptions
  - *is not callable* : n'est pas une fonction
  - *has no attribute...* : mauvais nom de méthode ou opérateur non supporté

## Ajouter du code pour débuser le problème

- `print()`, `sys.stderr.write()`
- Module *logging*

## Débogueur pas à pas

- `pdb` en ligne de commande
- `pdb` se pilote graphiquement avec des IDE comme *PyCharm* ou *Eclipse* avec l'extension *PyDev*

# Tests unitaires

## Plusieurs modules et outils

- *unittest* qui fait partie de la bibliothèque standard
- *pytest* qui est disponible sur pypi
- *nose* est un *fork* populaire de *pytest*

## Exemple avec *nose*

```
from yourmodule import price_of
def test_price_of_food():
 p = price_of('ham')
 assert p == 42
 p = price_of('spam')
 assert p == 12
 p = price_of('notfood')
 assert p == 0
 p = price_of('ham',10)
 assert p == 42 - 42/10
```

# Lancer les tests avec *nose*

## Il suffit d'exécuter *nosetests*

```
$ nosetests test_priceof.py
....

Ran 4 test in 0.001s
OK
```

## On peut obtenir un rapport de couverture du code

```
$ nosetests --with-coverage \
 --cover-html \
 test_priceof.py
$ firefox cover/index.html &
```

# Exercice final : Espion contre Jules César

## Contexte

Dans *La Guerre des Gaules* Jules César décrit le système de chiffrement utilisé par les armées romaines : il consiste à décaler d'un nombre entier, de façon circulaire les lettres de l'alphabet, les autres caractères restent inchangés.

## Problème

Nous avons intercepté un message secret utilisant sans doute un stratagème similaire, et nous savons qu'il contient le mot *spam*. Il nous faut le déchiffrer !

## Voici le message secret

Z UF EFK CZBV JGRD RK RCC!!! Z UF EFK NREK KYRK!  
A'VJGVIV HLV MFLJ MFLJ VKVJ RDLJVJ GVEUREK TV TFLIJ...

Quelle est le message envoyé par les romains ?

# Étapes pour briser le code

## Décomposez le problème

- Créez une fonction `encode_char(c,k)` renvoyant le caractère `c` chiffré par la clef `k` si `c` est un lettre, inchangé sinon
- Créez une fonction `encode_string(msg,k)` renvoyant la chaîne `msg` chiffrée par la clef `k`
- Créez une fonction `decode_string`

## Force brute

- Dans une boucle tentez de décoder le message secret avec la clef `k` parcourant l'intervalle `range(1,26)`
- Si la chaîne 'spam' fait partie du message : *bingo* ! on peut afficher le résultat et sortir de la boucle (*break*)
- Un *else* en fin de boucle exécute du code seulement dans le cas où on est pas sorti par un *break*

# Programmation Orienté Objet

## Python est intrinsèquement orienté objet

- Tous les types sont des classes
- Tout est un objet en Python: 42, *int*, *str*, fonctions, etc.

## Classes et instances

- Un type est une classe, par exemple *str*
- Une instance est un objet de ce type, 'spam'
- Le type est un *constructeur* : *str*(42)
- Nous pouvons créer nos propres classes

## Héritage

- Une classe peut *hériter* d'une autre
- Y compris les classes intégrées de Python, ou définies dans des modules
- La classe fille *hérite* des propriétés de la classe parente
- Et peut les redéfinir ou en définir de nouvelles

# Définition d'une classe

## L'instruction class

```
class City:
 '''Représentation d'une ville'''
 # Constructeur, appelé par City(...)
 def __init__(self,name):
 self.name = name
 def show(self):
 print(self.name)
```

## Un appel à la classe invoque le constructeur

```
paris = City('paris')
print(paris.name)
paris.show()
```

# Attributs et méthodes

Les attributs d'instance sont généralement positionnés par le constructeur

- Comme toutes les méthodes il reçoit l'objet en premier argument
- On le nomme typiquement *self*
- D'autres méthodes peuvent modifier ou ajouter des attributs

Les méthodes sont définies au niveau de la classe

- Reçoivent toutes *self* en premier argument
- Peuvent modifier les attributs d'instance
- Peuvent renvoyer un résultat

# Héritage

Une classe fille hérite des propriétés de la classe parente

```
class Capital(City):
 def __init__(self, name, country):
 self.country = country
 super().__init__(name)
 def show(self):
 super.show(self)
 print('Capitale de :', self.country)
```

*super()* permet d'appeler, si besoin, les méthodes de la classe parente

```
paris = Capital('Paris', 'France')
paris.show()
Capital.show(paris)
```

# Méthodes spéciales

## Les méthodes spéciales sont doublignées (*dunder methods*)

- Appelées implicitement
- `__init__` à la construction d'une instance
- `__str__` quand convertit en chaîne (par `print()` par exemple)
- `__add__` quand additionné à autre chose
- etc.

## Exemple avec `__str__`

```
class Capital(City):
 ...
 def __str__(self):
 return 'Capital({}, {})'.format(self.name, \
 self.country)

paris = Capital('Paris', 'France')
print(paris)
```

# Merci !

## Merci d'avoir suivi cette formation

- Nous espérons que vous vous êtes amusés à écrire des programmes Python
- ... Et que vous en écrirez de nombreux dans l'avenir

## Que nous reste-t-il à explorer ?

- La programmation objet en Python : créer nos propres classes !
- Le développement Web avec *Flask*, *Django*, ...
- L'accès aux bases de données *PostgreSQL*, *Sqlite*, *MySQL*, ...
- Le développement d'interfaces graphiques *Tkinter*, *Gtk*, *Qt*, *wxWindows*
- Le développement de jeux avec *PyGames*
- ... et bien plus encore !

