Éric Guichard

Python pour littéraires

Février 2025

Ouvrage au format 17 × 26 cm.
Version temporaire.
Illustration : abbaye de Jumièges.
Photo de l'auteur.

Licence : CC BY-NC-SA 4.0, soit :

— citation de l'auteur ;
— pas d'utilisation commerciale ;
— partage dans les mêmes conditions.

`https://creativecommons.org/licenses/by-nc-sa/4.0`

Préliminaires

But du livre

L'apprentissage de la programmation n'est jamais confortable car les machines ne pensent pas. Nous devons nous adapter aux simulacres de raisonnement que nous y avons implémentés, au prix de difficiles théorèmes. De plus, `python` est désarçonnant, même pour les personnes maîtrisant d'autres langages. Il faut « rentrer dedans », parfois apprendre sans comprendre, se remémorer certains « tics de langage », peut-être temporaires : il est possible que `python` soit obsolète dans dix ans.

D'un autre côté, il devient indispensable de savoir naviguer dans l'espace des signes informatiques et des formes associées. Un nouvel univers de l'écrit se déploie depuis une cinquantaine d'années et les personnes qui ne font que l'affleurer ou le consommer sans s'y plonger réellement finissent par disposer d'une faible littératie et donc par subir un réel handicap : un peu comme les personnes du 17^{e} siècle en Europe qui baignaient dans l'oralité. Elles pouvaient être très compétentes, elles avaient conscience que seuls les « lettrés » (celles et ceux qui savaient lire et écrire) décidaient : des valeurs morales, esthétiques, politiques qu'elles propageaient dans la société. Et s'il y avait des conflits de valeurs, ils se réglaient entre ces lettrés.

Aussi nous faut-il apprendre à écrire. Autant commencer par le « langage de programmation » le plus populaire du moment : si vous voulez apprendre les langues étrangères, il est plus profitable (et plus aisé) de commencer par l'anglais que par le basque ou une langue nilotique.

Le projet premier de ce guide est de vous *mettre le pied à l'étrier*, avec `python` : de vous aider à comprendre, après lecture, les manuels en ligne ou imprimés, et d'y choisir les solutions qui vous satisferont. J'ai essayé de faire simple : avec des définitions parlantes accompagnées d'exemples et de mises en perspective.

Son projet secondaire est de vous initier à l'algorithmique, au-delà de `python` : de vous familiariser avec des tours de main propres à l'infor-

matique, avec des invariants de la programmation (donc propres à tous les langages, comme les expressions régulières structurées par Schützenberger) et enfin, avec la rigueur, si proche de la conceptualisation. Une liste n'est pas un tableau (un dictionnaire en `python`), tous les nombres n'ont pas le même statut, un chiffre n'est pas un mot, même s'il existe parfois des passerelles permettant de transformer l'un en l'autre. Vous serez alors plus à l'aise s'il vous faudra aborder un autre langage ou quand vous voudrez « penser » l'informatique et la technique.

Il y a quelque rapport entre l'informatique et le jonglage textuel : un lien avec la littérature et la pensée plus dense qu'on ne le croit ; du fait que nous sommes les fabricants de nos instruments (c'est l'ordinateur qui imite le cerveau et non l'inverse) ; et aussi car ces machines sont fondées sur des raisonnements : des théorèmes puissants qui permettent des correspondances inimaginables pour le commun des mortels, ou qui garantissent des impossibilités éternelles. Par exemple, un ordinateur n'a accès qu'à un pourcentage infime (en fait nul) de tous les nombres existants : les « nombres calculables » sont dénombrables, l'ensemble des réels ne l'est pas. Or un ordinateur ne sort pas du champ du *calculable*.

J'ai privilégié la simplicité et une certaine forme d'universalité. C'est pour cela que des notions diverses (environnement, module, *package* et librairie) sont absentes ou peu abordées. Vous aurez toujours le temps, plus tard, de vous familiariser avec. Inversement, vous trouverez des pistes pour compter des mots [1], lire des fichiers « mal fichus » (95% des fichiers existants ?), faire vos premières enquêtes et produire les graphiques qui faciliteront vos analyses, traiter des fichiers en ligne, etc.

Contexte et précautions

Cette notice accompagne divers cours tenus à l'université de Lyon devant des étudiants n'ayant jamais programmé ou de culture dite « littéraire » : portés sur les mots et la langue. Elle est donc empiriquement efficace. Elle ne constitue pas à proprement parler une documentation pour `python`. Pour cela, vous pourrez consulter divers sites web. Beaucoup sont décevants, quelques uns très bons, comme ceux détaillés ci après. N'hésitez pas non plus à solliciter les « intelligences artificielles », souvent très pertinentes pour vous familiariser avec `python` [2].

1. La loi de Zipf est particulièrement approfondie. C'est la seule exception un peu dense dans ces ouvrage fondamentalement accessible. Mais elle est aussi « amusante ».

2. Par la suite, `python` sera parfois écrit simplement python.

Quelques références Liste non exhaustive.

— `https://docs.python.org/fr`,
— `https://coursinfopdf.com/introduction-python-3-pdf` (Bob Cordeau),
— `https://python.sdv.u-paris.fr/cours-python.pdf` ;
— pour les graphiques, je conseille le site `https://courspython.com` de David Cassagne.

D'autres références sont aussi indiquées dans le fil du texte.

Choix pédagogique

Il y a diverses manières d'aborder python. Par les *IDE* (Environnement de Développement Intégré) comme *Anaconda*, *via* des éditeurs dédiés comme *Visual Studio Code*, *Pycharm* ou *Spyder*, enfin en écrivant de petits programmes (*scripts*) compilés en ligne de commande (avec un *terminal*).

Selon les usages, elles ont leurs avantages ou inconvénients. Par exemple, Anaconda est lourd, *Pycharm* est vite payant, la troisième solution serait moins conviviale. Mais elle a des qualités :

— elle s'intègre parfaitement dans les précédentes ;
— elle est très efficace en dernier recours et
— parfaitement adaptée pour une initiation.

Aussi est-ce celle qui a été choisie pour cette documentation. Mais rien ne vous empêche d'intégrer les exemples présentés dans des outils des catégories ascendantes.

Mise en pratique

Note Dans les *scripts* de ce document, vous rencontrerez souvent l'apostrophe typographique `’` au lieu de `'`. Exemple : `a=’jour’`. Si vous les copiez, n'oubliez pas de remplacer ce symbole par l'apostrophe simple : `a='jour'`. Sinon, votre programme ne fonctionne pas.

Script Un *script* est un fichier texte (donc produit avec un éditeur de texte, lequel peut être l'un des précités).

```
mot="bonjour"
nombre=23
print (mot, nombre+1)
```

Il est recommandé de lui donner une extension `.py` et un nom. Appelons-le `pgm1.py`. Il se « lance » ainsi :

`python pgm1.py`

Le terminal affichera alors : `bonjour 24`

Aide pour écrire de longues lignes

Si vous devez écrire des longues commandes qui vous semblent illisibles sur une seule ligne, vous avez la possibilité de les scinder sur deux lignes avec un « \ ». Commençons par ce qui ne fonctionne pas :

```
print ("tot
","tat")
```

renvoie un `SyntaxError: EOL while scanning string literal`.

Alors que

```
print ("tot\
","tat")
```

ne produit pas d'erreur de syntaxe et renvoie ce que vous attendez : `tot tat`.

Il m'arrivera d'utiliser cette solution, par exemple à parti du point 2.3.3. Vous pourrez toujours supprimer ce \ et le saut de ligne qui le suit pour une lecture plus confortable.

Installations et précautions

La question des installations est assez bien documentée en ligne. Pour Mac et Windows, le site `https://www.python.org/downloads/?lang=fr` vous guide bien.

Si ce n'est déjà fait (*via* une installation directe), je conseille aussi d'installer `pip`, le gestionnaire de paquets pour Python.

Ensuite, sollicitez `pip` pour installer la bibliothèque de votre choix. Exemple, pour *Beautiful Soup* : `pip install beautifulsoup4`.

De façon générale, je conseille l'usage du *terminal* (ou de *Powershell*) pour lancer des commandes d'installation et de vérification.

Chapitre 1

Les bases

1.1 Variables, listes, dictionnaires, fonctions

Le « langage » `python` propose diverses catégories d'objets, résumées pour la pédagogie à trois : les variables, les listes, les dictionnaires. Mieux vaut leur donner des noms explicites :

`mot="soleil"` est mieux que `x="soleil"`.

Affirmons, par métaphore, qu'une liste est une succession de variables et qu'un dictionnaire est un tableau indexé : une sorte de couple de listes, la première composée de *clés*, associées à des *valeurs* qui constituent la seconde liste. Vous pouvez faire le lien avec `perl`, langage pour lequel une variable commence par un `$`, une liste par une `@` et un *hash* (tableau indexé) par un `%`.

1.1.1 Variables

Une variable peut être du texte (une chaîne de caractères), un nombre (entier ou à virgule : « flottant »), une chose de type vrai/faux (« booléen », apparenté à 1 / 0).

Un roman entier peut être inséré dans une variable, que l'on peut transformer en liste : par exemple de mots, délimités par la ponctuation. Exemples :

```
phrase="Il fait beau ce matin"
abreviation-mois="03"
nombre=3
```

Une variable textuelle peut être prédéfinie en étant mise à vide : `phrase=""`. De même pour une variable numérique, mise à zéro : `nombre=0`.

1.1.2 Commentaires, variables et caractères bizarres, etc.

Apostrophes et guillemets

En anglais : *single and double quotes* : ' et ".

Absence de différences Une variable texte peut indifféremment être détaillée entre guillemets ou entre apostrophes :

`a="bonjour"` ou `a='bonjour'`

Remarque J'aurais dû écrire `a='bonjour'` mais cela me prend beaucoup plus de temps. D'où la **Note** au début du document.

Dans la pratique, on essaie d'utiliser le délimiteur qui est le moins utilisé.

Si j'écris `a='aujourd'hui'`, j'induis une erreur : il y a 3 apostrophes. Je préfèrerai alors écrire `a="aujourd'hui"`.

Par exemple, si j'ai besoin d'insérer dans une variable l'expression
`<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink= "http://www.w3.org/1999/xlink"`
j'écrirai `a='<svg xmlns="http...xlink"'` car il n'y a pas d'apostrophes dans cette expression pleine de guillemets.

Une autre solution consiste à *protéger* l'apostrophe (ou les guillemets) du mot : `a='aujourd\'hui'`

Cas des variables avec plusieurs lignes

Deux solutions :

— avec des sauts de ligne encodés :
`a='première ligne\nseconde ligne'` ou
`a="première ligne\nseconde ligne"`.

— en utilisant 3 guillemets : `a="""un paragraphe séparé par de banals retours chariots"""` produit le résultat escompté. Cette solution fonctionne aussi avec 3 apostrophes.

Note Le langage `perl` n'aurait pas eu le même comportement : il interprète ce qui est entre guillemets, mais pas ce qui est entre apostrophes : `"\n"` produit un saut de ligne, `'\n'` produit un `\` suivi d'un `n`.

Longs commentaires Comme python ne tient pas compte des formes textuelles non assignées à une variable, tout paragraphe entre 3 apostrophes ou guillemets sera pris comme un commentaire (détournement fréquent) :

```
''' Ici
un long
commentaire'''
```

1.1.3 Additions et surprenantes concaténations

Selon les contextes, le signe `+` symbolise l'addition entre deux nombres ou la concaténation de deux chaînes de caractères. Exemple.

```
mot="bonjour"
nombre=23
print (mot+mot, nombre+nombre)
```

Réponse : `bonjourbonjour 46`

Si j'écris `a="bonjour"` et `b=" à tous"` `print (a+b)` donne donc `bonjour à tous`.

Mais si j'écris `c=2`, un `print (a+b+" "+c)` me renverra un message d'erreur, alors que si j'avais écrit `c="2"`, j'aurais obtenu comme imaginé `bonjour à tous 2`.

Il vous faut convertir `c` en chaîne de caractères pour l'intégrer en une autre (`string`) : `print (a+b+" "+str(c))`

1.1.4 D'une chaîne à un nombre

L'inverse peut se produire quand on lit une série dans un fichier : ce qu'on croit être un nombre `n` pourra être pris pour un caractère et on ne pourra alors pas l'utiliser pour des calculs.

En ce cas, la fonction (pour une conversion en entier) est `int()`. Ex. : `int(n)`. Pour une conversion en nombre banal (décimal) : `float(n)`. Pour s'y retrouver, la fonction `type` précise le type de la chose manipulée :

`print (type(a))` donne `<class 'str'>`

`print (type(c))` donne `<class 'int'>`

`print (type(2.5))` donne `<class 'float'>`

`liiiste=[2,5]` suivi d'un `print (type(liiiste))` donne `<class 'list'>`.

1.1.5 Extrapolation

Soit la série d'instructions suivantes, séparées par des lignes.

`e="2"  f=5  g="."  h=e+g+str(f)  print (h)  i=float(h)`

`print (type(i/2))` donne `<class 'float'>`. Ouf !

Afficher un résultat Cela s'effectue avec la fonction `print` :
`print(phrase)` donne `Il fait beau ce matin`
`print("Le jour")` affiche `Le jour`.

Vous **abuserez** de cette commande pour vérifier que votre programme fonctionne comme vous le voulez.

1.1.6 Listes

Une liste mise à vide s'écrit `liste=[]`. On dit qu'elle est alors « initialisée ». Listes et dictionnaires doivent souvent être initialisés avant usage.

On peut définir une liste de façon exhaustive :
`liste=[2,3,"le soir",18, "hier"]`.

Les éléments d'une liste sont numérotés à partir de **zéro** ; et ça « boucle » un peu :
`print (liste[2])` donne `le soir`.
`print (liste[-1])` donne `hier`.

Toute liste a une longueur (ici 5) : `len(liste)`. Il s'ensuit que le dernier élément de la liste est `liste[len(liste)-1]`.

Auto-indexation de listes : `enumerate`

Cf. le pt 1.2 pour la syntaxe des boucles.

Un exemple valant mieux que de longs discours...

```
for index, element in enumerate(liste):
        print ("l'index auto de l'élément <",element,"> vaut",index)
```

Résultat :

```
l'index auto de l'élément < 2 > vaut 0
l'index auto de l'élément < 3 > vaut 1
l'index auto de l'élément < le soir > vaut 2
l'index auto de l'élément < 18 > vaut 3
l'index auto de l'élément < hier > vaut 4
```

La variable `index` est donc créée automatiquement et donne la valeur courante du rang de l'élément sur lequel on travaille.

Une autre façon de voir l'usage de cette fonction consiste à demander le résultat de `print (list(enumerate(liste)))`.

La machine répond :
`[(0, 2), (1, 3), (2, 'le soir'), (3, 18), (4, 'hier')]`.

Listes inattendues

Les variables textuelles sont d'étranges listes (de caractères) à leur façon :
`phrase="Il fait beau ce matin."`
`print (phrase[3])` renvoie `f` (le 4e caractère).

Et donc `print (liste[-1][2])` renvoie `e`.

Étrangement, `print (phrase[3:5])` renvoie `fa` (2 éléments seulement : le 5e caractère est *exclu* : cf. point 1.2.1).

Ceci **ne fonctionne pas pour les (variables) nombres**, qui ne sont pas découpables en listes :
`a="123"` # ici `a` est une chaîne de caractères.
`print (a[0])` renvoie `1`. Mais
`b=123` # ici `b` est un nombre.
`print (b[0])` renvoie `TypeError: 'int' object is not subscriptable`.

Compléter une liste

Ajouter un élément à une liste : avec la fonction `append`. Ex. : `liste.append(18)`. Un `print (liste)` donne alors
`[2, 3, 'le soir', 18, 'hier', 18]`.

Cet exemple prouve qu'une liste peut contenir plusieurs éléments identiques, à la différence des clés d'un tableau-dictionnaire.

Fusionner deux listes

Utiliser l'opérateur de concaténation : `+`.

Exemple : soit une nouvelle liste : `laste=[19,20]`.
`print (liste + laste)` donne :
`[2, 3, 'le soir', 18, 'hier', 18, 19, 20]`

Réduire une liste : pop

Enlever le 4e élément de notre liste (complétée) :

```
liste.pop(3)
print (liste)
```

On obtient : `[2, 3, 'le soir', 'hier',18]`

On peut aussi garder en mémoire l'élément enlevé :

```
a=liste.pop(0)
print ("liste: ",liste," et la variable a vaut:",a)
```

Réponse :

```
liste:  [3, 'le soir', 'hier', 18] et la variable a vaut: 2
```

`pop(0)` est souvent utilisé pour enlever (et mémoriser) la première ligne d'un fichier de données (celle qui contient les noms des variables).

Supprimer le dernier élément de la liste Deux syntaxes :

- `liste.pop[-1]` clair : on enlève le dernier élément
- `liste=liste[:-1]` cf. la commande `range` pt 1.2.1 : on conserve tous les éléments de la liste jusqu'à celui du dernier indice, donc on perd le dernier élément de la liste (*no comment...*)

Fabriquer une liste à partir d'une variable : split

`split(sep)` coupe la variable en rondelles selon le séparateur `sep` (qui disparaît). Exemple banal : `a="le chat dort le soir au coin du feu"`

```
loste=a.split(" ")
print (loste)
```

donne

```
['le', 'chat', 'dort', 'le', 'soir', 'au', 'coin', 'du', 'feu']
```

Autre exemple, plus farfelu mais instructif

```
laste=a.split("o")
print (laste)
```

donnera

```
['le chat d', 'rt le s', 'ir au c', 'in du feu']
```

Cette commande est très utilisée, notamment avec les expressions régulières. Vous en trouverez divers usages dans cette documentation. Par exemple dans le programme de comptage de mots (point 2.1.4).

Note : les *whitespaces*, souvent traduits par « espaces » en français, sont en fait des « caractères blancs » : espace ␣, tabulation ou saut de ligne.

Pour utiliser les *whitespaces* comme séparateurs, deux solutions :

— utiliser `split` sans attribut : `laste=a.split()` ou
— expliciter ces séparateurs (`\s`) : `laste=re.split("\s+",a)` (après avoir importé le module `re` : c'est un *autre* « `split` » qui est ici sollicité).

Le `+` de `\s+` signifie : apparaissant une ou plusieurs fois. Cf. aussi le point 2.1.1.

Fabriquer une variable à partir d'une liste

Cas fréquent : copier une (fraction de) liste dans un fichier. On désire que cette liste soit facile à (re)traiter : on la munit de séparateurs.

```
chaine='\t'.join(liste)
```

Si la liste est hétérogène, cela ne fonctionne pas :

```
TypeError: sequence item 0: expected str instance, int found
```

Il vous faut convertir les nombres en chaîne de caractères. Utilisez alors la commande `map`, qui applique la fonction sollicitée (ici `str`) à toute la liste :

```
chaine='\t'.join(map(str, liste))
```

Un `print (chaine)` donne alors
`3  le soir  18  hier` (sachant que le précédent `liste.pop[-1]` a enlevé le dernier élément).

1.1.7 Dictionnaires ou tableaux

Sous sa forme simple, un dictionnaire est une succession de couples du type `clé, valeur`, comme des prénoms auxquels on associe l'âge des personnes évoquées. Un dictionnaire peut aussi être un emboîtement de dictionnaires élémentaires (analogue à des fichiers `json`).

Un dictionnaire initialisé s'écrit `dict={}`.

Évidemment, toutes les clés d'un dictionnaire sont uniques.

Diverses manières de remplir un dictionnaire

```
dict={"abc":2, "bcd":3}
dict["abc"]=2   etc.
```

Expliciter le contenu d'un dictionnaire

```
for cle in dict.keys():
    print("à la clé", cle, " est associée la valeur ", dict[cle]))
```

Autre **façon** :

```
for cle, valeur in dict.items():
    print("à la clé", cle, " est associée la valeur ", valeur)
```

Changer une ou plusieurs valeurs Élémentaire : `dict["abc"]=5` suivi de `dict["g"]="la nuit"` modifie la valeur de la clé `abc` et ajoute la clé `g`.

Retrouver les valeurs d'un dictionnaire Pour vérifier que tout fonctionne comme vous le désirez, vous pouvez solliciter un `print (dict.values())` Réponse : `dict_values([5, 3,'la nuit'])`

Récursivité La valeur d'une clé peut être une variable (cas précédents), une liste ou même un dictionnaire. Un dictionnaire est confortable quand on le produit soi-même. S'il est « donné » sans que sa structure soit explicitée (cas de certains fichiers `json` importés), la reconstitution de son squelette peut s'avérer complexe.

1.1.8 Fonctions

Une fonction est l'équivalent d'une routine en `perl`. Elle est composée de variables, et plus largement d'« objets » qu'il faut parfois initialiser. Elle se termine par un `return chose`, où `chose` est le résultat qu'on désire. Exemple simple.

```
def add (a,b):
        return (a+b)
```

Ainsi, `add(2,3)` renverra 5.

Une fonction peut être assez longue. Vous en trouverez un exemple au point 2.1.3.

fonctions *lambda*

Python propose un type particulier de fonctions, dites *anonymes*, qui permettent d'écrire du code bref, souvent adapté aux commandes `sorted()`, `map()` et `filter()`. Leur syntaxe est `lambda arguments: résultat`.

Par exemple, la fonction précédente aurait pu s'écrire `addit = lambda x,y: x + y`. Et un `print (addit(2,3))` aurait (aussi) donné 5.

Trier un dictionnaire avec une fonction anonyme

Un exemple courant consiste à trier un dictionnaire élémentaire par ses valeurs. Si
`tt={"a":2768, "e":6860, "i",3564,"s",4050}`, un
`tttrie = dict(sorted(tt.items(), key=lambda truc: truc[1], reverse=True))`
suivi d'un

```
for cle in tttrie.keys():
    print (cle, tttrie[cle])
```

renverra

```
e       6860
s       4050
i       3564
a       2768.
```

La clé de tri est le second élément d'une *chose* (que j'ai appelée *truc*) qui fait référence à ce qui précède : `tt.items()`, dont nous savons que c'est un couple `(clé, valeur)`. Le tri s'effectuera donc par les valeurs (le second élement, d'indice 1). `reverse=True` permet de produire un tri de la plus grande valeur à la plus petite.

1.2 Boucles et conditions

Nous les avons déjà rencontrées. Détaillons la syntaxe des `for`, `if`, `while`, etc. Syntaxe générale :

```
ligne de la condition:
-- > (TAB)   actions si condition réalisée
```

En d'autres termes, la ligne du type `for`, `if`, `else`, `elif`, `while` (je déconseille ce dernier aux débutants) se termine par deux points « : ». Les lignes correspondantes commencent par une **tabulation**.

1.2.1 Cas élémentaire : listes ordonnées et `for`

`range(0,10)` contient dix nombres, de **0 à 9**. Le dernier (10) est donc **exclu**. Preuve-exemple :

```
liste=range(0,10)
for i in liste:
        print (i)
        print ("Attention "+str(len(liste))+" n'est pas affiché")
```

S'affichent `0, 1`, etc. jusqu'à `9` et la ligne `Attention 10 n'est pas affiché`.

1.2.2 Sorties de boucles ou conditions

On usera de 3 types d'instructions : `continue` (sortie temporaire), `break` (sortie définitive) et `pass` (peu utile).

Exemples

```
for i in liste:
        if i==3: #si i vaut 3
                break
        print (i)
```

Seront affichés : 0,1,2.

```
for i in liste:
        if i==3:
                 continue
        print (i)
```

Seront affichés : 0,1,2, puis 4, 5… 9.

```
for i in liste:
        if i %2==0: #si i divisé par 2 donne un reste nul
                print (str(i)+ " est pair")
        else:
                pass
```

Dans ce dernier cas, les lignes contenant `else` et `pass` sont en fait inutiles. Mais elles peuvent être confortables

— si vous doutez de vous (ex. : pas de `if` sans `else` ?),

— si vous voulez remettre à plus tard les instructions du `else`.

1.2.3 Opérateurs conditionnels

Je nomme ainsi les façon d'exprimer les opérateurs logiques `et`, `ou`, etc. Voici les plus utilisés. Nous avons vu que « si i vaut 3 » s'écrit `if i==3` (avec deux signes « = »). Cela vaut aussi pour des chaînes de caractères : `if  dict["g"]=="la nuit":`

Différent de : `if` Exemple : `if  dict["g"] !="la nuit": ...`

Cela peut aussi s'écrire
`if not dict["g"]=="la nuit":`

Dans une liste `in` Exemple :

```
if "soir" in liste:
        print("c'est bon")
else:
        print ("souci!")
```

Renvoie `souci!` : c'est l'expression `le soir` qui est dans la liste, et non pas le mot `soir`.

Combinaisons : and, or Exemple utilisant le pt 2.1.1 :

```
if dict["g"].endswith("nuit") and dict["g"].startswith("la"):
        print ("ok")
```

Un `ok` s'affiche.
Note : Les mots clés `and` et `or` peuvent être remplacés par les symboles `&` et `|`.

Autres : <, etc. Exemple `if 3<=4:` `if "jour" <= "nuit":` (dans ce contexte, `<` équivaut à « avant »).

Attention à `if 3<="la nuit":`, qui crée une erreur : un nombre ne se compare pas à une chaîne de caractères.

1.2.4 Compléments sur les conditions

elif Nous avons vu l'alternative `if... else`. Il est possible de la décomposer sous des formes du type `si... sinon si... sinon` autant de fois que désiré (pour les `sinon si`) même si on s'y perd vite. Syntaxe et exemple :

```
if  dict["g"]==3:
        print ("la valeur de g est ",dict["g"])
elif dict["g"]==4:
        print ("la valeur de g est ",dict["g"])
else:
        print ("la valeur de g n'est ni 3 ni 4")
```

La réponse donnée est : `la valeur de g n'est ni 3 ni 4`.

1.2.5 Compter des objets

En `python`, vous *ne* pouvez *pas* créer de tableaux (dictionnaires) à la volée comme en `perl`, même si la fonction `enumerate` a ses charmes

(cf. point 1.1.6). Il faut donc les *initialiser*. Un tel cas fréquent consiste à compter tous les mots d'une liste et leurs fréquences.

Exemple : Voici une liste de lieux de naissance de quelques personnes. `liste=[Paris,Lyon,Paris,Rouen,Bron,Lyon]`. Combien sont nées à Paris, etc. ? Voici une solution, qui sollicite les boucles et conditions.

```
liste=["Paris","Lyon","Paris","Rouen","Bron","Lyon"]
sommelieux={}
for v in liste:
        if v in sommelieux.keys(): # si on a déjà vu v
                sommelieux[v]+=1 #on ajoute 1 à sa valeur associée
        else:
                sommelieux[v]=1 # on crée la clé v et on lui associe 1
print (sommelieux)
```

Ce qui donne : `{'Paris': 2, 'Lyon': 2, 'Rouen': 1, 'Bron': 1}`

Vous pouvez remplacer la dernière ligne par celles-ci :

```
for i in sorted(sommelieux.keys()):
        print (sommelieux[i],"personne(s)  née(s) à", i)
```

Résultat :

```
1 personne(s) née(s) à Bron
2 personne(s) née(s) à Lyon
2 personne(s) née(s) à Paris
1 personne(s) née(s) à Rouen
```

1.3 Lire un fichier, écrire dans un fichier

Pour manipuler un fichier, usez de la commande `open`, à nuancer au fil de vos compétences. Dans les cas simples, le fichier est ouvert en « mode lecture » (`r` comme *read*), sinon en « mode écriture » (`w` comme *write*), si vous comptez le créer et y inscrire des « choses ».

1.3.1 Exemple de lecture simple

```
fic=open("UNfichierExistant","r")
# fic est l'objet python correspondant au fichier du disque dur
lig=list(fic)
print (type (lig)) # Réponse: <class 'list'>
print (lig[1])     #renvoie la seconde ligne du fichier
....
fic.close() # n'oubliez pas de fermer le fichier virtuel
```

Vous pouvez alors travailler sur les lignes, qui sont des éléments de la liste `lig`. Ici `lignecourante` est un nom de variable mnémotechnique.

```
fic=open("UNfichierExistant","r")
lig=list(fic)
for lignecourante in lig:
        print (lignecourante)
        ...
```

1.3.2 Écrire simplement

`monfic=open("NouveauFic", "w")` crée (ou écrase !) `NouveauFic`.

Pour y inscrire la variable `a`, il suffit d'écrire `monfic.write(a)`. Mais **attention** : `a` doit être une chaîne de caractères. Abusez au besoin des `str(a)`.

Pour écrire dans un fichier existant *sans le supprimer*, c'est-à-dire pour *compléter* ce fichier : choisir l'option `"a"` pour *append*. Exemple : `monfic=open("UnFichier", "a")`.

1.3.3 Ouverture conventionnelle d'un fichier

Les documentations font état d'une syntaxe plus efficace, qui rend inutile la fermeture du fichier sollicité, mais qui induit une tabulation de plus :

```
with open("cerclesbiz", "r",encoding='utf8') as fic:
        lig=list(fic)...
```

De façon générale, précisez toujours l'**encodage**. Ex. : `encoding='utf8'`.

1.3.4 Exemple de lecture avec `csv.reader`

`reader` est un « objet » du module `csv`, qui facilite la lecture de fichiers tabulés.

Soit le fichier `cercles` qui a l'allure suivante et dont nous voulons récupérer les coordonnées X, Y des centres :

```
X       Y       R
10      90      5
30      60      10
60      80      15
```

La 5e ligne du programme suivant présente un usage de `pop(0)` (cf. le point 1.1.6).

```
import csv
nomInterneDuFichier = csv.reader(
open("cercles", 'r', encoding='UTF-8'), delimiter='')
totaldeslignes = list(nomInterneDuFichier)
totaldeslignes.pop(0) # si vous ne voulez pas de la 1re ligne

for lignecourante in totaldeslignes:
        print (lignecourante)
        coord1,coord2,rien=lignecourante #grâce au délimiteur
        #la variable inutile (rien) est indispensable...
        #autre solution: coord1=lignecourante[0], etc.
        print (coord2) #à compléter...
```

Note Il est aussi possible de lire des fichiers *formatés* par des logiciels externes (ex. : XL ou LibreOffice) avec d'autres *packages* ou modules (ex. : `openpyxl` ou `pandas`).

1.3.5 Autres situations de lecture

Lire un fichier tabulé mal construit

Que faire quand le fichier tabulé n'est pas régulier ? Cf. le fichier `cerclesbiz` :

```
X        Y        R
10        90        5        7
30        60        10        2        2
60        80        15
90        45
```

La solution consiste à mettre chaque ligne dans une liste (nommée ici `rien`) et à récupérer les variables présentes.

```
fic=open("cerclesbiz","r")
lig=list(fic)
rien=[]
for lignecourante in lig:
        rien=lignecourante.split('\t') # liste de longueur variable
        x=rien[0]
        y=rien[1]
        print (x,y)
```

Vers les fichiers du web

Un exemple simplissime, pour donner le goût du traitement des fichiers en ligne.

```
import requests
url = "https://www.ericguichard.fr"
web = requests.get(url)
contenu=web.text
if "designers" in contenu:
        print ("ok")
else:
        print ("aie")
```

La machine répond « ok » : le mot « designers » est bien dans la page html appelée.

Lire une série de fichiers

Faites usage du module `os` : *operating system*. Exemple.

```
import os
os.chdir("1erTourbis") # on va dans ce dossier
print(os.listdir()) #affiche son contenu
for fichier in os.listdir(): #pour chaque fichier
        if fichier.endswith("ZZ"): #si le nom du fichier finit par ZZ
                with open(fichier, "r",encoding='utf8') as circons:
                        lignes=list(circons)
                        for chaqueligne in lignes:
                                ...
```

1.4 Formatage de nombres et commande `format`

Cf. `https://courspython.com/print-format.html?highlight=format` et `https://python.sdv.univ-paris-diderot.fr/03_affichage`.

1.4.1 Usage des chaînes formatées : *fstring*

Il s'agit en général d'obtenir des expressions simplifiées ou homogènes de valeurs numériques. Exemple, déclinable de façons très variées :

```
a=2
b=3.1278
print(f"a vaut {a:.2f} et b vaut {b:.2f} ")
```

Résultat : `a vaut 2.00 et b vaut 3.13`

Ce formatage est indépendant de la fonction `print` :
`c=f"{b:.2f}"` # n'oubliez pas les guillemets !
`print (c)` donnera `3.13`.

1.4.2 la fonction format

Exemple simplissime

```
print ("x vaut {}, y vaut {} et z vaut {}".format('2','3','4'));
```

Donne comme résultat
`x vaut 2, y vaut 3 et z vaut 4.` Très utile pour le svg...

Second exemple

```
res = ['papier', 'carton', 'chemise', 'encre']
print("Nous avons de l'{} et du {} en stock\n".format(res[3],res[0]))
#autre solution, moins lisible:
print("Nous avons de l'{0[3]} et du {0[0]} en stock\n".format(res))
```

Voilà pour les choses élémentaires relatives à python.

Chapitre 2

Textes, graphiques et rudiments de statistiques

Dans ce chapitre sont abordés les rudiments de nettoyages textuels, jusqu'à l'usage de motifs (*patterns*) efficaces via les expressions rationnelles (*regexp*). Vous apprendrez à produire le dictionnaire d'un roman.

Ensuite viennent les graphiques élémentaires et leur application à la loi de Zipf ; suit une initiation aux statistiques (tris à plats et croisés de variables issues d'un tableau d'enquête) ; le point sur les graphiques est alors repris pour vous aider à analyser vos résultats. Pour étayer vos conclusions, sont présentés les moyens de réaliser le test du Khi-2.

Enfin, j'explique comment python dialogue avec d'autres logiciels.

2.1 Nettoyages textuels, expressions régulières

2.1.1 Premiers nettoyages

Fonctions simples

- **`replace`** La fonction `replace` est fort confortable. Exemple, sachant qu'`\n` est un saut de ligne :

```
phrase="Il fait beau\nce matin"
phrase=phrase.replace('\n'," ")
print (phrase)
```

Résultat : `Il fait beau ce matin`.

- **strip** `strip` est utile quand il s'agit de travailler sur des objets comme des noms de fichiers qu'on veut débarrasser des espaces (␣) qui parfois les enveloppent. Ex. :
  `a="␣fichier␣␣"`
  `a=a.strip()`
  `print ("T"+a+"T")` renvoie `TfichierT`.

  Paramètres complémentaires :

  `strip('liste de caractères bordants à enlever')` Ces caractères peuvent être dans le désordre mais tous doivent être présents (y compris l'espace au besoin).

Attention `strip` enlève les *whitespaces* (cf. point 1.1.6). En bref, pour `a="Essai␣␣ \n"` (des espaces, des tabulations et un saut de ligne à droite), `a.strip()` renverra un `Essai` tout court.

Seconde remarque La gestion des espaces (et des sauts de ligne) par `python` peut s'avérer désarçonnante. Par exemple, `print ("XXX","YYY")` affiche un `XXX YYY` (introduction d'une espace).

- **Commence ou finit par...** Les commandes `endswith` et `startswith` sont simples et utiles.

```
a="coucou"
if a.endswith("u"):
print (a)
```

Renvoie `coucou`.

Jongler avec les capitales, les caractères accentués...

Que faire si, sur 1000 lignes, vous avez des formes graphiques du type « PRÉNOM NOM » que vous voulez transformer en « Prénom Nom » ?

- **Solution globalement satisfaisante :**

```
nom="PRÉNOM NOM"
nom=nom.lower()
nom=nom.title()
print (nom)
```

Renvoie `Prénom Nom`.

Détournement pour capitales complexes Vous devinez rapidement comment écrire une capitale comme « Ç » :
`print ('française'.upper())`, suivi d'un copier-coller et le tour est joué.

- **Solution générale** `casefold()` est préféré à `lower()` car il réduit plus de capitales que ce dernier. Ainsi, `print ('ß'.casefold())` donne un `ss`. L'inverse de `casefold()` n'existant pas, vous pourrez, dans de telles situations compliquées, jouer avec des `replace` ou fabriquer des dictionnaires de conversions. Exemple :
`dico = {"grosses": "großes", "essen": "eßen"}`.

2.1.2 Vers les expressions « régulières »

Ce terme est un anglicisme, mieux traduit en français par « expression rationnelle ». Ses abréviations peuvent être *regex* ou *re*, d'où le nom du module associé de python.

Il s'agit d'attraper des « motifs » textuels (*patterns*) un peu complexes : toutes les expressions d'un roman entre guillemets, tous les mails de vos collaborateurs, tout ce qui ressemble à un nombre décimal, *et cetera*. Souvent, les expressions rationnelles servent à nettoyer un texte. C'est un outil puissant, présent dans la plupart des « langages de programmation » ; nous le décrivons **très** brièvement ici.

Substitution : `sub`

Exemple simple.

```
import re
a="bonjour"
b=re.sub("jou","soi",a)
print (b)
```

Renverra `bonsoir`.

Ci dessous, on débarrasse le texte de sa ponctuation, qui sera mise entre crochets. Le « + » signifie qu'au moins un des caractères entre crochets doit être présent. En général, un caractère spécial (comme le point « . ») doit être protégé par une contre-oblique (*backslash*).

```
import re
lignecourante="Le terme « poésie » et ses dérivés « poète »,
« poème » viennent du grec ancien (poiesis)"
```

```
lignesimple=re.sub('[ \.,«»()\']+', ' ', lignecourante)
print (lignesimple)
```

Réponse : `Le terme poésie et ses dérivés poète poème viennent` du `grec ancien poiesis`

Chaînes dites brutes Pour éviter de multiplier les protections (les contre-obliques : \), vous pouvez insérer un « r » devant votre expression régulière. Mais comme vous désirez remplacer l'apostrophe par une espace, il vous faudra la protéger... sauf si vous insérez votre expression rationnelle en des guillemets. En d'autres termes, les 3 lignes suivantes produisent le même résultat. À noter que des outils comme *Spyder* peuvent accepter ou refuser des protections \ selon les *préférences*.

```
lignesimple=re.sub('[ \.,«»()\']+', ' ', lignecourante)
lignesimple=re.sub(r'[ .,«»()\']+', ' ', lignecourante)
lignesimple=re.sub(r"[ .,«»()']+", ' ', lignecourante) #idéale
```

Recherche : `search`

Cherchons tous les mots entre guillemets. Une première idée serait d'utiliser la *regex* (abrev. d'expression rationnelle) suivante : `'«.+»'` : guillemet ouvrant, puis n'importe quel caractère apparaissant au moins une fois, puis guillemet fermant. Essayons, avec une syntaxe un peu modifiée et avec la fonction `search`. Nous serons déçus...

```
motif='«.+»'
test= re.search(motif,lignecourante)
if test: # si le motif est trouvé
        print (test) # voyons à quoi ça ressemble
        resu=test.group()
        print ("Résultat: ",resu)
```

Réponse : `<re.Match object; span=(9, 55), match='« poésie »` et ses dérivés `« poète », « poème »'>`
`Résultat: « poésie » et ses dérivés « poète », « poème »`

La machine nous « dit » que `resu` est un « *match object* » qui va du caractère 9 de `lignecourante` jusqu'au caractère 55, et explicite ce contenu en dernière ligne. Les expressions régulières sont « gourmandes » : la nôtre a tout attrapé, du premier guillemet ouvrant au *dernier* fermant.

Solution Précisons que nous désirons une forme graphique qui commence par un guillemet ouvrant, qui contient ensuite n'importe quel

caractère *sauf* un guillemet fermant et se termine par un guillemet fermant : `'(«[^»]+)»'`

```
motif = '«[^»]+»'
test = re.search(motif, lignecourante)
if test:
        resu = test.group()
        print(f"résultat: {resu}") #chgt de syntaxe, pour voir...
```

Et nous lisons : `résultat: « poésie »`. C'est mieux, mais nous n'avons que le premier terme. Pour les attraper tous...

Recherche complète : `findall`

```
motif = '«[^»]+»'
resu = re.findall(motif, lignecourante)
print (resu)
```

Et nous lisons : `['« poésie »', '« poète »', '« poème »']`. Cela devient plaisant. Vous l'avez compris, `resu` est une liste. Et donc `resu[0]` renvoie `« poésie »`.

Construction d'une liste *via* une expression rationnelle

Si par exemple vous cherchez tous les mots d'un texte, vous voudrez les « découper » selon la ponctuation, etc. La démarche est analogue au cas précédent, même si le motif est un peu plus développé :
`motif = r"[()«» ,.;'-]+"` le *tiret* doit être le dernier terme !

la commande `liste=re.split(motif, lignecourante)` vous produit alors la liste suivante (avec un `print (liste)`) :
`['Le', 'terme', 'poésie', 'et', 'ses', 'dérivés', 'poète', 'poème', 'viennent', 'du', 'grec', 'ancien', 'poiesis', '']`

Une solution plus simple consiste à utiliser comme motif tout ce qui n'est pas une lettre, un chiffre ou le caractère `_` :
`motif=r'\W+'` (`\W` : tout ce qui n'est pas un « word »).

Vous aurez remarqué qu'un mot vide termine la liste. La solution pour l'éliminer est un peu étrange :
`liste = [mot for mot in liste if mot]`

Elle renvoie à la syntaxe
`[expression for item in iterable if condition]`. Ici `if mot` revient à vérifier que l'élément `mot` de la liste n'est pas vide. Cette solution sera utilisée au point 2.2.4.

Voisinages de la « capture » avec search

Il peut arriver que vous cherchiez à savoir ce qu'il y a avant ou après ce que vous avez attrapé avec votre expression rationelle. La solution peut apparaître un peu technique, mais renvoie aux indices des listes, que vous connaissez déjà.

```
motif = '«[^»]+»'
test = re.search(motif, lignecourante)
if test:
        resu = test.group()
        avant= lignecourante[:test.start()] #jusqu'à résu
        apres= lignecourante[test.end():] # jusqu'à la fin
        print ("on a trouvé:", resu)
        print ('il y avait avant:', avant)
        print ('et ensuite:', apres)
```

S'affiche alors :

```
on a trouvé: « poésie »
il y avait avant: Le terme
et ensuite: et ses dérivés « poète », « poème » viennent du
grec ancien (poiesis)
```

Autres voisinages et mises en mémoire

Cette méthode est proche de la précédente, mais un peu plus technique. Elle est assez « universelle » et c'est pourquoi je l'apprécie.

Supposons que seul le premier mot entre guillemets vous intéresse et que vous vouliez garder en mémoire ce qui précède le premier guillemet ouvrant et ce qui suit le dernier fermant. Cet exemple étrange a essentiellement pour fonction de vous montrer les vertus du parenthésage dans les *regex*. Idée :

1. trouver d'abord ce qu'il y a avant le 1er guillemet ouvrant : `[^«]+`
2. ensuite, notre premier mot entre guillemets : `«[^»]+»`
3. après, ce qu'on veut jusqu'au dernier guillemet fermant : `.*»`
4. enfin, ce qui suit jusqu'au dernier caractère : `.*$`

Pour information, le `$` signale la fin de la forme graphique. Si vous mettez certaines expressions entre parenthèses, elle seront prêtes à être mémorisées : à entrer dans une variable. Ce qui est fait ici, avec les *regex* 1, 2 (sans ses guillemets) et 4.

Étrangement, c'est une des rares fois où les indices informatiques commencent à 1 et non à 0.

```
motif='([^«]+)«([^»]+)».*»(.*$)'
test = re.search(motif, lignecourante)
if test:
        resu = test.group()
        print ("capture:", resu)
        print ("R1:",test.group(1)) #1re parenthèse. Oui, 1 et non 0!
        print ("R2:",test.group(2)) #2e, etc.
        print ("R3:",test.group(3))
```

Résultat :

```
capture: Le terme « poésie » et ses dérivés « poète », « poème »
viennent du grec ancien (poiesis)
R1: Le terme
R2: poésie
R3: viennent du grec ancien (poiesis)
```

Vous comprenez qu'on peut glisser `test.group(1)`, etc. dans des variables *ad hoc*.

2.1.3 Application aux fonctions

Cette partie prolonge le point 1.1.8.

Voici un exemple de fonction qui nécessite une initialisation : la fonction `comptagemots` s'applique à une liste, elle renvoie un dictionnaire dont les clés sont les éléments de la liste (*a priori* des mots) et les valeurs leur fréquence.

```
def comptagemots(liste):
        totalm={}
        for m in liste:
                if m in totalm:
                        totalm[m]+=1
                else:
                        totalm[m]=1
        return totalm
```

2.1.4 Premier comptage des mots d'un texte

Nous allons utiliser tout ce que nous savons jusqu'ici et utiliser la fonction précédente. Pour éviter un affichage trop long des résultats, nous nous limitons aux mots apparaissant au moins deux fois. Notez aussi la condition `len(i)==0... continue` pour régler le souci des mots vides.

```
import re
def comptagemots(liste):
        totalm={}
        for m in liste:
                if m in totalm:
                        totalm[m]+=1
                else:
                        totalm[m]=1
        return totalm

fichier=open("VotreFichier", "r") #votre fichier de travail
leslignes = list(fichier)
totalmots={}
tousmots=[]
motif = r"[() .,;«»()'\n-]+"

for lignecourante in leslignes:
        mots = re.split(motif, lignecourante)
        tousmots+=mots        #concaténation de listes!

totalmots=comptagemots(tousmots)
fichier.close()        #fin du travail de comptage

for i in totalmots:
        if (len(i)==0) | (totalmots[i] <2): # > 2 apparitions
                continue
        print (i," apparaît ",totalmots[i]," fois")

print (sorted(totalmots.keys())) # ordre bizarre: Z devant é
```

2.2 Rudiments graphiques

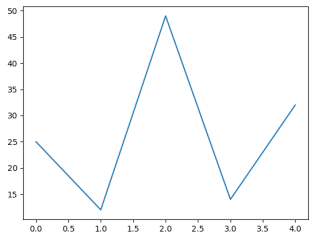
Les exemples suivants donnent une idée des possibilités de `matplotlib`.

2.2.1 Listes de nombres

C'est le cas le plus simple. Pour représenter une série de points de coordonnées (x, y), vous fabriquez la liste des x, notée ici `abscisses` et celle des y, notée ici `ordonnees`. Les graphiques de la figure 2.2 page 36 sont ainsi construits. La syntaxe générale est alors :
`plt.plot(abscisses, ordonnees)`. Exemple :

```
import matplotlib.pyplot as plt
abscisses=range(0,5)
```

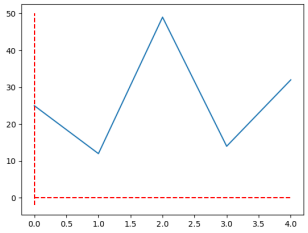
```
freq = [25, 12, 49, 14, 32]
plt.plot(abscisses, freq)
plt.show()                                    # Ne pas oublier!
```



Pour mieux voir les hauteurs exactes des points, vous pouvez demander (avec la même logique : `plt.plot`) à visualiser les axes :

```
import matplotlib.pyplot as plt
abscisses=range(0,5)
freq = [25, 12, 49, 14,32]
plt.plot(abscisses, freq)
plt.plot((0, 0), (-2, 50), "r--") #axe vertical
plt.plot((0, 4), (0, 0), "r--")
plt.show()
```

Vous obtenez alors le graphique

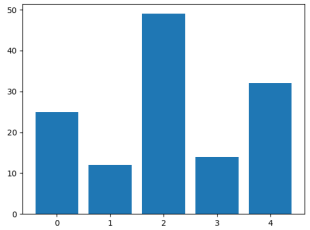


2.2.2 Histogrammes

Il suffit de remplacer `plot` par `bar`.

```
...
plt.bar(abscisses, freq)
plt.show()
```

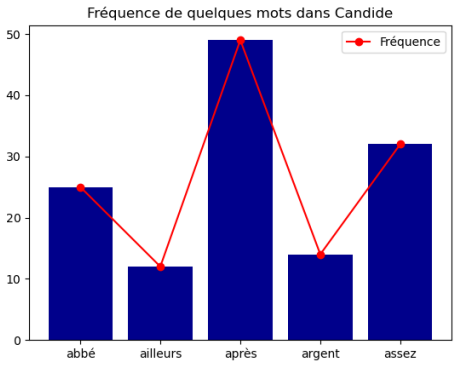
Ce qui donne :



Cette méthode fonctionne aussi pour des catégories non numériques (histogrammes généraux). Vous pouvez jouer à l'infini sur la présentation (titres, couleur, etc.).

```
import matplotlib.pyplot as plt
mots= ['abbé', 'ailleurs', 'après', 'argent','assez']
freq = [25, 12, 49, 14,32]
plt.bar(mots, freq,color='darkblue') #ou plot
plt.plot(mots, freq,"o-", label='Fréquence',color='red')
plt.title("Fréquence de quelques mots dans Candide")
plt.show()
```

Voici le résultat (deux représentations différentes pour les mêmes données) :



Le point 2.3 vous permettra d'approfondir la question des graphiques multiples.

2.2.3 Graphes de fonctions

C'est assez simple. Exemple.

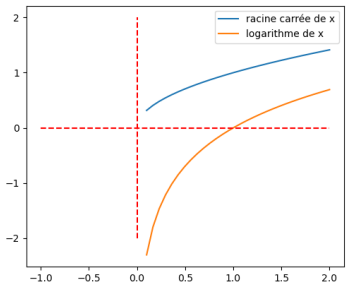
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0.1, 2, 30)
#fabrique une liste de 30 éléments entre 0.1 et 2
y1 = np.sqrt(x) #racine de x
y2 = np.log(x)  #log de x

plt.plot(x, y1, label="racine carrée de x")
plt.plot(x, y2, label="logarithme de x")

plt.plot((0, 0), (-2, 2), 'r--') #repérage axes
plt.plot((-1, 2), (0, 0), 'r--')
plt.legend()
plt.show()
```

Ce qui donne :



2.2.4 Visualisation de la loi de Zipf et hapax en couleur

Un hapax est un mot qui n'apparaît qu'une fois dans un texte. De tels mots sont nombreux (environ la moitié du dictionnaire d'un roman) mais « pèsent » peu : 10% du total des fréquences dans le *Candide* de Voltaire. À l'opposé, certains mots sont très fréquents, comme les articles.

La distribution des mots dans un (long) texte ne renvoie pas aux lois dotées d'une variance (où les marges pèsent peu), mais aux lois « de puissance ». On appelle « loi de Zipf » la distribution *rang* (en abscisse) et fréquence *en ordonnée* de ces mots.

Son graphe ressemble à une hyperbole. Rapporté à des échelles loga-

Comment Candide fut élevé dans un beau château, et comment il fut chassé d'icelui.
Il y avait en Vestphalie, dans le château de M. le baron de Thunder-ten-tronckh, un jeune garçon à qui la nature avait donné les moeurs les plus douces. Sa physionomie annonçait son âme. Il avait le jugement assez droit, avec l'esprit le plus simple; c'est, je crois, pour cette raison qu'on le nommait Candide. Les anciens domestiques de la maison soupçonnaient qu'il était fils de la soeur de monsieur le baron et d'un bon et honnête gentilhomme du voisinage, que cette demoiselle ne voulut jamais épouser parce qu'il n'avait pu prouver que soixante et onze quartiers, et que le reste de son arbre généalogique avait été perdu par l'injure du temps.
Monsieur le baron était un des plus puissants seigneurs de la

Fig. 2.1 – Copie d'écran du résultat html : les hapax sont en rouge foncé, la ponctuation en vermillon et les mots dont la fréquence dépasse 1% du total en bleu (les couleurs du programme ont été changées car le bleu initial se voyait peu). Le programme affiche : `Il y a 35494 mots dans ce texte, dont 5491 différents, incluant 3121 hapax.`

rithmiques, il est quasiment linéaire : comme une droite de pente -1. Le programme qui suit (proche de celui du point 2.1.4) visualise ces deux graphes. Il produit aussi (et d'abord) un fichier html qui met en vert les mots très fréquents, en rouge les hapax et en bleu les séparateurs visibles. Les IAs proposent des solutions courtes pour un tel traitement, mais au-delà du niveau de cette documentation ; et par ailleurs parfois incomplètes et lentes.

La méthode choisie repère les mots et leurs séparateurs, compte les mots, puis les balise avec des drapeaux (un `XXX` devant et un `YYY` derrière). Ensuite, est effectué un découpage par ces drapeaux, qui permettra le coloriage. Le résultat est produit figure 2.1. Il faut être prudent avec les débuts et fins de lignes : les séparateurs peuvent être vides.

```
import re
import math
import matplotlib.pyplot as plt

def comptagemots(liste):
        totalm={}
        for m in liste:
                if m in totalm:
                        totalm[m]+=1
                else:
                        totalm[m]=1
        return totalm

introhtml='''<html lang="fr">\n<head>
<meta http-equiv=content-type content="text/html;
charset=utf8"/> \n</head>\n<body>\n'''
finhtml='''\n</body>\</html>'''

# Premières initialisations et paramètres
fichier= open('Candide', "r",encoding='utf8')
ficsortie=open('Hapaxcouleur.html', "w",encoding='utf8')

totalmots={}
ensemblesep={}
listemots=[]
tousLesSeparateurs=[]
roman=""
nblignes=0
suite=""
tailletexte=0

motif="(\w+)([\W_-]*)(.*$)" # attention à l'ordre _- et non -_
# motif= mot, séparateur, suite de la ligne:
# tous mis en mémoire grâce aux parenthèses
leslignes = list(fichier)
```

```
for lignecourante in leslignes:
        final=""
        phr=lignecourante
        while phr !="":
                mots = re.findall(r'motif', phr.lower())
                motpris=""
                test = re.search(motif, phr)
                if test:
                        motpris=test.group(1)
                        tailletexte+=1
                        listemots.append(motpris.lower())
                        suite=test.group(3)
                        separ=test.group(2) #Repérage sépar...
                        ensemblesep[separ]=1
                        totalmots[separ]=0
                        tousLesSeparateurs.append(separ)
                final+="YYY"+motpris+"ZZZ"+separ
                phr=suite
        roman+=final+"\n<br>" # pour le html
fichier.close()

totalmots=comptagemots(listemots)
hapax = [mot for mot in totalmots.keys() if totalmots[mot]==1]

#Autres initialisations
roman2=""
decoupZ=[]
motif2=r"[\W_-]+"
totalhapax=0

listetrucs=roman.split("YYY")
for m in listetrucs:
        decoupZ=m.split("ZZZ") # mot:[0], séparateur=[1]
        if len(decoupZ)==2: # évite les séparateurs vides
                ponct='<font color="blue">'+str(decoupZ[1])+'</font>'
        sansZ=decoupZ[0]
        if len(sansZ)==0: # si le mot est vide
                continue
        if totalmots[sansZ.lower()]==1: #hapax
                totalhapax+=1
                sansZ='<font color="red">'+sansZ+'</font>' # ou white!!!
        elif (totalmots[sansZ.lower()] > tailletexte/100):
                sansZ='<font color="green">'+sansZ+'</font>'
        roman2+=sansZ+ponct
        sansZ=""
        ponct=""

ficsortie.write(introhtml)
ficsortie.write(roman2)
ficsortie.write(finhtml)
ficsortie.close() # fin html
```

```
print ("Il y a ",tailletexte,"mots dans ce texte, dont")
print (len(totalmots),"différents, incluant", totalhapax,"hapax")

#Graphique données brutes
listenb=sorted(totalmots.values(),reverse=True)
rangs=range (1, len(listenb)+1)
plt.plot(rangs, listenb,label="données rang/fréquence brutes")
plt.legend()
plt.show()

#Graphique log-log
leslogsy=[]
for j in listenb:
        logy=math.log(j)
        leslogsy.append(logy)
        leslogsabs=[]
for j in rangs:
        logx=math.log(j)
        leslogsabs.append(logx)
plt.plot(leslogsabs, leslogsy,label="données rang/fréquence log/log")
plt.legend()
plt.show()
```

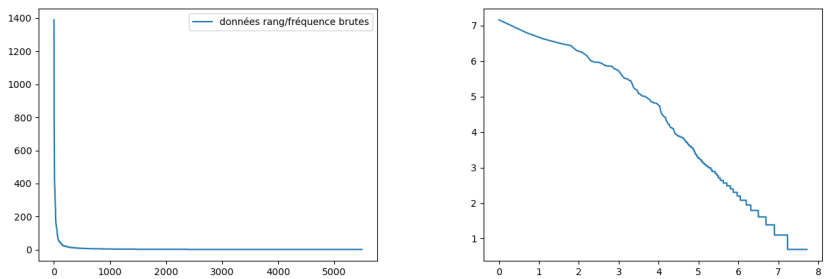


Fig. 2.2 – À gauche, graphique brut de la loi de Zipf réalisé par le programme (couple *rang-fréquence* des mots). Il est quasiment illisible. À droite, le graphique *log-log* ; il a une allure linéaire, de pente environ -1.

Ce programme et le fichier `Candide` sont disponibles à l'URL `https://www.ericguichard.fr/python/datapgm`.

2.3 Statistiques élémentaires

Nous nous plaçons ici dans le registre des enquêtes avec variables qualitatives, et donc des fichiers (texte) ayant cette allure :

```
Nom         Variable1         Variable2
Dupont         H                 bleu
Durand         F                 vert
Dubois         H                 orange...
```

Il s'agit de décrire sommairement les variables (pourcentages, histogrammes, etc.), de les croiser, de vérifier si elles sont indépendantes : test du χ^2 (ou Khi-2 : *chi-square*), comparaison (graphique) de profils, etc. : le b-a-ba du traitement d'enquête.

2.3.1 Tris à plat « à la main »

La plupart des opérations peuvent se réaliser avec les méthodes précédentes. Par exemple, pour produire l'histogramme d'une variable, il vous suffit

— d'ouvrir le fichier (avec `csv.reader` s'il est tabulé, cf. pt 1.3.4),
— de transformer la colonne de la variable en liste (cf. pt 1.1.6),
— de compter les éléments de cette liste (*via* un dictionnaire, cf. pt 1.2.5),
— puis de réaliser l'histogramme de ses clés et valeurs (point 2.2.2).

2.3.2 Tris à plat avec `pandas`

Pour cette documentation élémentaire, nous considérons que `pandas` fabrique et manipule des tableaux de données (*dataframes*) et des séries, même si ce *package* fait bien plus que cela. Le programme suivant donne comme réponse (exemple tiré d'un fichier pédagogique [1]) : les femmes sont 2685 et représentent 54% des individus de notre fichier.

```
import pandas as pd
data = pd.read_csv('Votrefichier.csv')
#ajouter delimiter="\t" si l'extension n'est pas .csv
modalites=data["Variable1"].value_counts()
modalitesENpc=data["Variable1"].value_counts(normalize=True)
print (modalites, modalitesENpc)
```

1. Ce fichier, nommé `TB1`, est aussi disponible à l'URL `https://www.ericguichard.fr/python/datapgm`.

Réponse :

```
Variable1
F    2685
H    2243
Name: count, dtype: int64 Variable1
F    0.544846
H    0.455154
Name: proportion, dtype: float64
```

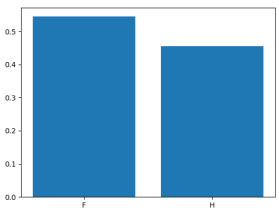
Dans le script, `data` est un *dataframe* et `modalitesENpc` une série. Une série est une (sorte de) liste caractérisées par son *index* et son *array*, ce qui la rend proche d'un dictionnaire. En l'occurrence,

`print (modalitesENpc.index)` donnera un `Index(['F', 'H'], dtype='object', name='Variable1')` et un `print (modalitesENpc.array)` produira trois lignes dont celle-ci : `<PandasArray> [0.5448457792207793, 0.4551542207792208]`.

Il s'ensuit qu'un `plt.bar(modalitesENpc.index,modalitesENpc.array)` produira l'histogramme de notre variable (cf. point 2.2.2). `array` peut être remplacé par `values` mais non par `values()` (différence entre attributs et méthodes) :
`plt.bar(modalitesENpc.index,modalitesENpc.values)` produit le même graphique.

```
import pandas as pd
import matplotlib.pyplot as plt
data = pd.read_csv('Votrefichier.csv')
modalitesENpc=data["Variable1"].value_counts(normalize=True)
plt.bar(modalitesENpc.index,modalitesENpc.array)
plt.show()
```

Ce script de 6 lignes produit le calcul et le graphique de l'histogramme :



Rappel sur les csv `csv` signifie souvent que la virgule sert de séparateur de colonne. Si votre séparateur est autre, par exemple une tabulation (notée `\t`), il vous faudra le préciser dans le script :
`data = pd.read_csv('Votrefichier.csv',delimiter="\t")`

2.3.3 Tableaux croisés

Il est aussi aisé de les produire. L'option `normalize` ramène ces tableaux à l'état de pourcentages en colonnes (le total de chaque colonne vaut 100% : `normalize='columns'`) ou en ligne (le total de chaque ligne vaut 100% : `normalize='index'`). On parle alors (respectivement) de profils colonnes et de profils lignes.

```
import pandas as pd
data = pd.read_csv('Votrefichier.csv')
tableau_croise = pd.crosstab(data['Variable2'], data['Variable1'])
tableau_pcc = pd.crosstab(data['Variable2'], data['Variable1'],\
normalize='columns')
tableau_pcl = pd.crosstab(data['Variable2'], data['Variable1'],\
normalize='index')
```

Résultat (après un `print(...)`), toujours tiré du même fichier pédagogique, où la variable `Variable2` a 3 modalités : `1Xj`, `qqX` et `ssobj`.

Tableau des effectifs ↓

Variable1	F	H
Variable2		
1Xj	631	676
qqX	507	432
ssobj	1547	1135

Profils colonnes (pcc, comme pourcentage colonnes) ↓

Variable1	F	H
Variable2		
1Xj	0.235009	0.301382
qqX	0.188827	0.192599
ssobj	0.576164	0.506019

Profils lignes (pcl) ↓

Variable1	F	H
Variable2		
1Xj	0.482785	0.517215
qqX	0.539936	0.460064
ssobj	0.576808	0.423192

Filtrage

Vous pouvez « filtrer » vos données, par exemple si vous désirez éliminer des individus ambigus. Dans l'exemple qui suit, nous supprimons les personnes ayant répondu `qqX` (quelques usages de l'internet dans le mois précédant l'enquête).

```
data = pd.read_csv('Votrefichier.csv')
filtre=data['Variable2']!="qqX" # différent de qqX
datafiltree=data[filtre]
tableau_croise = pd.crosstab(datafiltree['Variable2'], \
datafiltree['Variable1'])
etc.
```

Vous pouvez combiner les filtres Ici, des (anciennes) régions sont regroupées :

```
filtreRiches=(datafiltree['REGION']=='RhoneAlpes') | \
(datafiltree['REGION']=='IdF')
```

Notez le \ qui rend lisibles des conditions complexes en permettant de les écrire sur plusieurs lignes (cf. les préliminaires). Avec de tels passages artificiels à la ligne, vous n'avez pas de souci de tabulation.

Recodages

Cf. aussi la page `https://www.sfu.ca/~mjbrydon/tutorials/BAinPy/05_recode.html`.

Une solution simple consiste à créer une variable à partir des filtres précédents, par exemple avec une fonction :

```
def recodReg(region):
        if (region =="IdF")| \
        (region =="RhoneAlpes")
                return "RegR"
        elif (region == "Auvergne") | \
        (region =="NordPdeC") | \
        (region =="Picardie") ):
                return "RegP"
        else:
                return "RegAmbigue"
```

Par ce biais, à partir d'une vingtaine de régions, nous en avons trois, insérées dans une nouvelle variable, nommée ici `typReg`. Pour qu'elle soit comprise par python (et pandas), il suffit d'un
`data['typReg'] = data['REGION'].apply(recodReg)`

Vous pouvez alors faire les tris croisés de votre choix. Exemple :
`tableausimpleR=pd.crosstab(data['Variable1'], data['typReg'])`
(évidemment, la chose peut s'opérer sur un *dataframe* préalablement filtré).

Graphiques des profils

Diverses solutions existent, parmi lesquelles la suivante :

Les lignes de nos tableaux s'obtiennent par leur identifiant, qui les *localisent*. Par exemple, `tableau_pcl.loc['1Xj']` donne

```
Variable1
F    0.482785
H    0.517215
```

Ce qui correspond bien au profil ligne de la modalité `1Xj`, même si on est surpris qu'il apparaisse comme une colonne. On obtient de même le profil des deux autres modalités. On peut alors les comparer graphiquement, comme on l'a déjà fait.

Ne manque plus que la légende (H, F). En fait, ce sont les indicateurs des colonnes. Nous pouvons alors tenter un `print(tableau_pcl.columns)`, qui nous donnera un `Index(['F', 'H']`, suivi d'autres choses.

Il nous suffit de reprendre la logique de la section 2.2.1, qui nous proposait d'oser un `plt.plot(abscisses, ordonnees)`. Or nous avons désormais ces deux listes d'abscisses et d'ordonnées :

```
plt.plot(tableau_pcl.columns, tableau_pcl.loc['1Xj'])
```

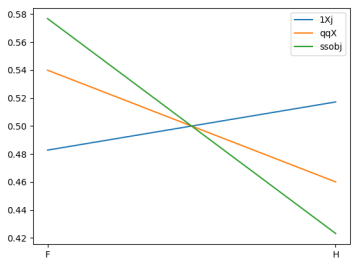
Il est alors aisé de comparer graphiquement les 3 profils lignes :

```
import pandas as pd
import matplotlib.pyplot as plt
data = pd.read_csv('Votrefichier.csv')

tableau_pcl = pd.crosstab(data['Variable2'], data['Variable1'], \
normalize='index')

plt.plot(tableau_pcl.columns, tableau_pcl.loc["1Xj"],label="1Xj")
plt.plot(tableau_pcl.columns, tableau_pcl.loc["qqX"],label="qqX")
plt.plot(tableau_pcl.columns, tableau_pcl.loc["ssobj"],label="ssobj")
plt.legend()
plt.show()
```

Vous comprendrez à l'usage pourquoi ce type de graphique est plus lisible qu'une somme d'histogrammes en bâtons :



Au premier regard, il semble que nos variables ne soient pas indépendantes : sinon les droites (les profils) auraient la même allure (et les mêmes pentes). En même temps, les échelles de pourcentage sont assez resserrées. Nous vérifierons cela au point 2.3.3 avec le test du χ^2 (Khi-2).

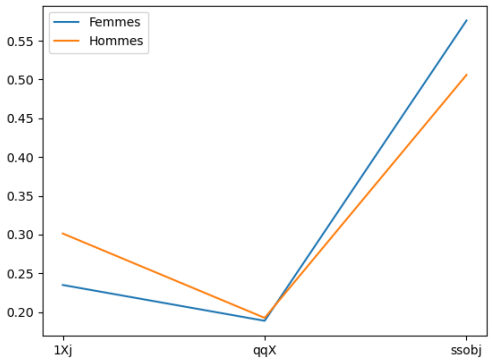
Nous pouvons vérifier notre intuition en considérant les profils des colonnes, afin de vérifier si les pratiques des femmes et des hommes diffèrent ou non. C'est plus simple que les pratiques des personnes `1Xj` !

Nous avons la légende : non pas `tableau_pcl.index`, mais (pour les colonnes) `tableau_pcc.index`. Pour mémoire, le tableau qui nous intéresse est décrit page 39 : celui des pourcentages colonnes (pcc). Pour obtenir le graphique associé, une façon rapide est de transposer le tableau et de faire le travail sur les lignes qu'on vient de faire (`tableau_pcl.T`). Une autre est d'inverser les deux variables. et la troisième est la suivante : attraper les colonnes par leur index (leur nom) : `tableau_pcc[["F"]]`.

Il s'ensuit qu'un `plt.plot(tableau_pcc.index, tableau_pcc[["F"]]` fera l'affaire. Et le complément suivant

```
plt.plot(tableau_pcc.index, tableau_pcc[["F"]],label="Femmes")
plt.plot(tableau_pcc.index, tableau_pcc[["H"]],label="Hommes")
```

produira ce graphique. Les différences de profil ne sont si pas excessives.



Test d'indépendance

Pour rester dans un cadre très général (le test du χ^2 ne s'explique pas en 30 secondes), nous dirons que l'appel `chi2_contingency(tableau_croise)` produit un ensemble de résultats dont les premiers sont

— la valeur du χ^2 total du tableau,
— la probabilité (souvent faible) qu'il ait cette valeur sous hypothèse d'indépendance,
— le nombre de degrés de liberté,

— et d'autres choses, qui commencent par `array` : le tableau des valeurs théoriques.

Si la probabilité est inférieure à $0,05$, nous rejetons « l'hypothèse d'indépendance » : il y a dépendance entre les deux variables. Exemple.

```
import pandas as pd
from scipy.stats import chi2_contingency
data = pd.read_csv('Votrefichier.csv')
tableau_croise = pd.crosstab(data['Variable2'], data['Variable1'])

print(chi2_contingency(tableau_croise))

print ("Khi-2 total:",chi2_contingency(tableau_croise)[0])
print ("Degrés de liberté:",chi2_contingency(tableau_croise)[2])
print ("On rejette l'hyp d'ind si ce nombre est inférieur à 0,05:",
chi2_contingency(tableau_croise)[1])
```

Et nous lisons, successivement, la valeur de `chi2_contingency(tableau_croise)` :

```
(31.439092345248675, 1.4896623422462532e-07, 2,
array([[ 712.11343344,  594.88656656],
[ 511.61018669,  427.38981331],
[1461.27637987, 1220.72362013]]))
```

puis les 3 réponses à nos questions (`print ("Khi-2 total, ...`) :

```
Khi-2 total: 31.439092345248675
On rejette l'hyp d'ind si ce nombre est inférieur à 0,05:
1.4896623422462532e-07
Degrés de liberté: 2
```

Comme 1.489×10^{-7} est bien plus petit que 0.05, nous rejetons sans souci l'hypothèse d'indépendance.

Attention Il faut faire le calcul sur le **tableau des effectifs** et non sur l'un des tableaux de profils lignes ou colonnes.

2.4 Dialoguer avec le monde

Python ne fait pas tout. C'est pourquoi il est utile de permettre à nos scripts d'appeler d'autres programmes. Nous avons vu un tel cas au point 1.3.5. Voici deux autres exemples proches qui permettent ce type de dialogue. Le premier vous permet d'insérer le résultat d'un programme dans une variable, le second de produire un nouveau fichier grâce à ce programme.

2.4.1 Le résultat d'un programme externe dans une variable

Il s'agit ici de récupérer les informations relatives à une photo. La commande désirée ici est `exiftool photo.JPG` (voir `https://exiftool.org`). Reconnaissons que la syntaxe de python n'est pas commode.

```
import subprocess

# Exécuter la commande et capturer la sortie
resu = subprocess.run(["exiftool", "photo.JPG"], \
stdout=subprocess.PIPE, text=True)

# La sortie est maintenant dans resu.stdout
a = resu.stdout
print(a)  # Affiche la sortie d'exiftool
```

La variable `a` contient désormais le résultat de notre commande `exiftool...`. Nous pouvons faire encore plus simple :

```
import subprocess
a = subprocess.check_output(["exiftool", "photo.JPG"], text=True)
print(a)
```

2.4.2 Un fichier produit par un programme externe

Ici, nous convertissons une photo JPG en GIF. La fonction `run` sollicite maintenant trois paramètres puisque la commande désirée est composée de trois termes : `convert photoDOrigine photoConvertie`. La solution suivante suffit car il est inutile de faire usage d'une variable :

```
import subprocess
subprocess.check_output(["convert", "photo.JPG","photo.gif"])
```

2.5 Pour conclure

Cette initiation terminée, vous saurez profiter des documentations et tirer profit des autres manières d'écrire et de lancer vos programmes (celles évoquées dans les préliminaires). N'oubliez pas que, comme d'autres « langages de programmation » (qui n'ont jamais été des langages), python renvoie à une culture : une culture de l'écrit, qui se complète d'une *façon d'appréhender le monde, de le partager, de s'y afficher*. C'est ici que l'informatique rejoint l'anthropologie : que cela devient passionnant.

Table des matières