Bases de programmation en Python

CPES DAC : Introduction à l'algorithmique

Pierre Senellart

12 septembre 2023

Plan

Algorithmique

- Vient du nom de محمد بن موسى الخوارزمى

Algorithmique

- Vient du nom de محمد بن موسى الخوارزمى (Muhammad ibn Musa al-Khwarizmi), un scientifique perse du IX^{e} siècle

Algorithmique

- Vient du nom de محمد بن موسى الخوارزمى (Muhammad ibn Musa al-Khwarizmi), un scientifique perse du IX^{e} siècle
- ... qui est aussi à l'origine du mot algèbre (الْجَبْر, *remise en place* (des os fracturés) en arabe), du titre de l'un de ses livres sur la résolution d'équations

Algorithmique

- Vient du nom de محمد بن موسى الخوارزمى (Muhammad ibn Musa al-Khwarizmi), un scientifique perse du IX^{e} siècle
- ... qui est aussi à l'origine du mot algèbre (الْجَبْر, *remise en place* (des os fracturés) en arabe), du titre de l'un de ses livres sur la résolution d'équations
- Un algorithme est une spécification formelle de la manière dont résoudre un problème donné : à partir d'une entrée, comment produire la sortie correspondant à la solution d'un problème, en combinant des opérations élémentaires

Algorithmique

- Vient du nom de محمد بن موسى الخوارزمى (Muhammad ibn Musa al-Khwarizmi), un scientifique perse du IX^{e} siècle
- ... qui est aussi à l'origine du mot algèbre (الْجَبْر, *remise en place* (des os fracturés) en arabe), du titre de l'un de ses livres sur la résolution d'équations
- Un algorithme est une spécification formelle de la manière dont résoudre un problème donné : à partir d'une entrée, comment produire la sortie correspondant à la solution d'un problème, en combinant des opérations élémentaires
- L'algorithmique est l'étude des algorithmes : conception de l'algorithme, analyse de sa performance, démonstration de sa correction, etc.

Algorithmique

- Vient du nom de محمد بن موسى الخوارزمى (Muhammad ibn Musa al-Khwarizmi), un scientifique perse du IX^{e} siècle
- ... qui est aussi à l'origine du mot algèbre (الْجَبْر, *remise en place* (des os fracturés) en arabe), du titre de l'un de ses livres sur la résolution d'équations
- Un algorithme est une spécification formelle de la manière dont résoudre un problème donné : à partir d'une entrée, comment produire la sortie correspondant à la solution d'un problème, en combinant des opérations élémentaires
- L'algorithmique est l'étude des algorithmes : conception de l'algorithme, analyse de sa performance, démonstration de sa correction, etc.
- La programmation est la manière de transformer (on dit implémenter) un algorithme en un code dans un langage informatique, afin d'exécuter l'algorithme sur des données concrètes

Algorithmique et programmation

- Tout algorithme est implémentable : il doit être décrit en des termes suffisamment précis pour qu'il n'y ait pas d'ambiguïté sur comment le transformer en programme
- ... mais cela ne veut pas dire que le programme implémentant l'algorithme est facile à écrire, car le programmeur doit prendre en compte les limitations de la machine, les particularités du langage, des objets de bas de niveau et pas des concepts de haut niveau, etc.
- Algorithme : abstraction de ce qui est implémentable
- Le langage de programmation n'a pas d'influence sur ce qui est implémentable ; tous les langages de programmation ont le même pouvoir d'expression (on dit qu'ils sont Turing-complets)
- ... mais un langage a un impact sur la facilité (cf. `https://pierre.senellart.com/other/languages/languages.xml`) ou sur l'efficacité de l'implémentation

Structure de données

- Élément de base utilisé dans des algorithmes plus complexes, réutilisé dans des algorithmes divers pour résoudre différents problèmes
- Spécification formelle d'un objet mathématique abstrait (liste, ensemble, graphe, matrice, etc.), des opérations possibles sur cet objet (insertion, énumération, inversion, etc.) et des algorithmes les réalisant
- Élément implémentable, souvent sous la forme d'une classe en programmation orientée objet
- Souvent possible de concevoir différentes structures de données pour le même objet mathématique, plus ou moins efficaces pour une tâche donnée

Pourquoi Python ?

- Beaucoup de langages de programmation, avec leurs forces et faiblesses
- Langage simple pour de petits programmes, et particulièrement adapté à la science des données
- Un des langages les plus populaires dans l'industrie, voir `https://survey.stackoverflow.co/2023/`
- Certain(e)s d'entre vous ont déjà fait du Python en lycée
- Mais attention aux spécificités :
  - La permissivité du langage (typage dynamique, non-déclaration de variables, etc.) rend plus difficile de détecter certains problèmes
  - Manque de certaines constructions d'autres langages (par exemple, en programmation orientée objet)
  - Parfois nettement plus lent que d'autres langages
  - Attention à l'indentation !

Plan

Qui fait quoi ?

- L'algorithmicien conçoit, décrit et analyse un algorithme

Qui fait quoi ?

- L'algorithmicien conçoit, décrit et analyse un algorithme
- Le programmeur implémente cet algorithme dans un langage de programmation donné, pour un environnement donné

Qui fait quoi ?

- L'algorithmicien conçoit, décrit et analyse un algorithme
- Le programmeur implémente cet algorithme dans un langage de programmation donné, pour un environnement donné
- Dans les langages interprétés (comme Python) :

Qui fait quoi ?

- L'algorithmicien conçoit, décrit et analyse un algorithme
- Le programmeur implémente cet algorithme dans un langage de programmation donné, pour un environnement donné
- Dans les langages interprétés (comme Python) :
  - le compilateur transforme le programme en bytecode spécifique au langage

Qui fait quoi ?

- L'algorithmicien conçoit, décrit et analyse un algorithme
- Le programmeur implémente cet algorithme dans un langage de programmation donné, pour un environnement donné
- Dans les langages interprétés (comme Python) :
  - le compilateur transforme le programme en bytecode spécifique au langage
  - l'interpréteur exécute le bytecode instruction par instruction

Qui fait quoi ?

- L'algorithmicien conçoit, décrit et analyse un algorithme
- Le programmeur implémente cet algorithme dans un langage de programmation donné, pour un environnement donné
- Dans les langages interprétés (comme Python) :
  - le compilateur transforme le programme en bytecode spécifique au langage
  - l'interpréteur exécute le bytecode instruction par instruction
- Dans les langages compilés (comme C) :

Qui fait quoi ?

- L'algorithmicien conçoit, décrit et analyse un algorithme
- Le programmeur implémente cet algorithme dans un langage de programmation donné, pour un environnement donné
- Dans les langages interprétés (comme Python) :
  - le compilateur transforme le programme en bytecode spécifique au langage
  - l'interpréteur exécute le bytecode instruction par instruction
- Dans les langages compilés (comme C) :
  - le compilateur transforme le programme en code assembleur spécifique à la machine

Qui fait quoi ?

- L'algorithmicien conçoit, décrit et analyse un algorithme
- Le programmeur implémente cet algorithme dans un langage de programmation donné, pour un environnement donné
- Dans les langages interprétés (comme Python) :
  - le compilateur transforme le programme en bytecode spécifique au langage
  - l'interpréteur exécute le bytecode instruction par instruction
- Dans les langages compilés (comme C) :
  - le compilateur transforme le programme en code assembleur spécifique à la machine
  - l'assembleur transforme le code assembleur en code machine directement exécutable par le processeur

Exemple d'algorithme

Entrée : Tableau T avec n éléments distincts, un élément x

Sortie : la position de x dans T

1: **for** $i \leftarrow 0$ **to** $n - 1$ **do**
2: **if** $T[i] = x$ **then**
3: **return** i
4: **end if**
5: **end for**
6: **return** *pas trouvé*

Programme Python

```
def find(x):
  for i in range(0, n):
    if T[i] == x:
      return i
  return -1
```

Bytecode Python

```
          0 SETUP_LOOP              43 (to 46)
          3 LOAD_GLOBAL              0 (range)
          6 LOAD_CONST               1 (0)
          9 LOAD_GLOBAL              1 (n)
         12 CALL_FUNCTION            2
         15 GET_ITER
    >>   16 FOR_ITER                26 (to 45)
         19 STORE_FAST               1 (i)
         22 LOAD_GLOBAL              2 (T)
         25 LOAD_FAST                1 (i)
         28 BINARY_SUBSCR
         29 LOAD_FAST                0 (x)
         32 COMPARE_OP               2 (==)
         35 POP_JUMP_IF_FALSE       16
         38 LOAD_FAST                1 (i)
         41 RETURN_VALUE
         42 JUMP_ABSOLUTE           16
    >>   45 POP_BLOCK
    >>   46 LOAD_CONST               2 (-1)
         49 RETURN_VALUE
```

Programme C

```
long find(int x) {
  for(long i=0; i<=1000; ++i)
    if(T[i]==x)
      return i;
  return -1;
}
```

Code assembleur x86-64 (compilé depuis C)

```
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        mov     QWORD PTR [rbp-8], 0
.L5:    cmp     QWORD PTR [rbp-8], 1000
        jg      .L2
        mov     rax, QWORD PTR [rbp-8]
        mov     eax, DWORD PTR [rax*4+0x404060]
        cmp     DWORD PTR [rbp-20], eax
        jne     .L3
        mov     rax, QWORD PTR [rbp-8]
        jmp     .L4
.L3:    add     QWORD PTR [rbp-8], 1
        jmp     .L5
.L2:    mov     rax, -1
.L4:    pop     rbp
        ret
```

Code machine x86-64

```
        push    rbp                                 #55
        mov     rbp, rsp                            #48 89 e5
        mov     DWORD PTR [rbp-20], edi             #89 7d ec
        mov     QWORD PTR [rbp-8], 0                #48 c7 45 f8 00 00 00 00
.L5:    cmp     QWORD PTR [rbp-8], 1000             #48 81 7d f8 e8 03 00 00
        jg      .L2                                 #7f 1d
        mov     rax, QWORD PTR [rbp-8]              #48 8b 45 f8
        mov     eax, DWORD PTR [rax*4+0x404060]     #8b 04 85 60 40 40 00
        cmp     DWORD PTR [rbp-20], eax             #39 45 ec
        jne     .L3                                 #75 06
        mov     rax, QWORD PTR [rbp-8]              #48 8b 45 f8
        jmp     .L4                                 #eb 0e
.L3:    add     QWORD PTR [rbp-8], 1                #48 83 45 f8 01
        jmp     .L5                                 #eb d9
.L2:    mov     rax, -1                             #48 c7 c0 ff ff ff ff
.L4:    pop     rbp                                 #5d
        ret                                         #c3
```

Plan

Historique de Python

1991 Première version publique de Python, développée par Guido van Rossum

1994 Première version majeure (1.0) de Python, début de son adoption

2000 Python 2; de nombreux changements, Python 2 a été utilisé de manière intensive dans de nombreux projets logiciels

2008 Débuts de Python 3, incompatible avec Python 2; Python 3 a tardé à remplacer Python 2, longue période de cohabitation entre ces deux versions

2020 Python 2 n'est plus maintenu

On ne parlera que de Python 3. Mais on trouve toujours de temps en temps des programmes Python 2...

Programme Python

- Programme écrit sous la forme d'un (ou plusieurs) fichier(s) texte, éditables avec n'importe quel éditeur de texte (VS Code, emacs, vim, Bloc-Notes Windows, TextEdit...)
- En règle générale : une étape du programme, une instruction, s'écrit comme une ligne dans le fichier texte, avec retour à la ligne après ; possible de couper les lignes trop longues, mais seulement à certains endroits
- Contrairement à la plupart des langages de programmation, pas de caractère (tel que « ; ») terminant les instructions
- Tout ce qui suit un caractère « # », jusqu'à la fin de la ligne est ignoré : commentaire
- L'indentation (nombre d'espaces en début de ligne) sert à délimiter les blocs des structures de contrôle (pas de « { } » comme dans d'autres langages)
- Guides de style, cf. `https://peps.python.org/pep-0008/`

Jeux de caractères

Unicode : répertoire de caractères, assignant à chaque caractère, de quelque langue que ce soit, un entier

A	→	65	ε	→	949
é	→	233	$\aleph$	→	1488

Jeu de caractères : moyen de représenter concrètement, par une suite de 0 ou de 1, un caractère Unicode

Par exemple, pour le caractère « é » :

latin1	`11101001 (Seulement pour certains caractères)`
utf-8	`11000011 10101001`
utf-16	`11101001 00000000`

utf-8 présente l'avantage de pouvoir représenter tous les caractères d'Unicode, de manière compatible avec l'ancien encodage ASCII

En Python, les programmes sont des fichiers textes en utf-8 par défaut (et pas de bonne raison de changer ça)

Variables

- Dans un langage de programmation, une variable est un nom donné à un emplacement de la mémoire de l'ordinateur
- Une variable a :

  Un nom : qui permet de lui faire référence ; en Python commence par une lettre ou un tiret bas et contient lettres, tiret bas et chiffres (mais les noms commençant par des tirets bas sont réservés pour des usages spéciaux

  Un type : entier relatif, nombre réel avec un nombre de chiffres (binaires) et un exposant fixé (nombre à virgule flottante), suite finie de caractères (chaîne de caractères), booléen, etc.

  Une valeur : une valeur de ce type, stocké dans la mémoire associée à cette variable

Typage dynamique vs statique

- Deux grandes catégories de langages en fonction de comment ils traitent les variables :

  Typage statique : les variables ont un type affecté dès la première référence à cette variable (sa déclaration) ; ce type doit être explicitement déclaré (dans un langage comme C, Pascal) ou est automatiquement inféré du contexte (Haskell, OCaml, ou dans certains cas en Java, C++, Scala...)

  Typage dynamique : le type associé à une variable peut changer au fur et à mesure du temps, à chaque fois qu'une nouvelle valeur est associée à cette variable

- Python est un langage à typage dynamique ; de plus, les variables ne sont pas explicitement déclarées, elles existent à partir du moment où on commence à les utiliser

Typage dans différents langages

Comment créer une variable a de type entier et lui affecter la valeur 42 dans différents langages ?

```python
# Python: pas de déclaration,
# pas de typage statique
a = 42
```

```perl
# Perl: déclaration,
# pas de typage statique
my $a = 42;
```

```c
// C: déclaration,
// type déclaré
int a = 42;
```

```cpp
// C++: déclaration,
// type inféré
auto a = 42;
```

Types de base Python et littéraux

Type	Exemples de « littéraux »
NoneType	**None**
bool	**True** **False**
int	42 -1234567890123456789 0b1001 0xdeadbeef
float	3.14159 6.02214076e23
complex	1j -4.5+2j
str	"Hello" 'Bonjour'
bytes	b'\xc3\xa9'
list	[] [1] [1,2,3] ['a',1,3.14]
tuple	() (1,) (1,2,3) ('a',1,3.14)
set	{1} {1,2,3} {'a',1,3.14}
dict	{}, {'a':1, 'b': 2}
range	range(10) range(0, 10) range(0, 10, 1)

Opérateurs

Arithmétique : `+`, `-`, `*`, `/`, `//`, `%`, `**`

Comparaisons : `<`, `<=`, `>`, `>=`, `==`, `!=`, **`is`**, **`is not`**

Booléens : **`or`**, **`and`**, **`not`**

Appartenance : **`in`**, **`not in`**

La priorité des opérateurs est celle à laquelle on s'attend, mais cf. `https://docs.python.org/fr/3/reference/expressions.html#operator-precedence` pour les détails ; en cas de doute, utiliser des parenthèses !

Fonctions pré-définies

Python comprend beaucoup de fonctions pré-définies, comme :

`abs(nombre)` valeur absolue d'un nombre, ou module d'un nombre complexe

`int(valeur)` transforme une valeur en un entier (par exemple, en interprétant une chaîne de caractères comme un entier)

`len(collection)` nombre d'éléments d'une collection (par exemple, de type `list` ou `str`)

`max(valeur1, valeur2)` maximum de deux valeurs

`min(valeur1, valeur2)` minimum de deux valeurs

`str(valeur)` transforme une valeur en une chaîne de caractères

Instructions simples

- Une instruction est une étape élémentaire exécutée par l'interpréteur Python
- Instructions d'affectation
  - `var = val` affecte à la variable `var` la valeur `val`
  - `var += val` est équivalent à `var = var + val` ; marche aussi avec `*=`, `-=`, etc.
  - Attention : `=` est une instruction d'affectation, tandis que `==` est un opérateur de comparaison
- **pass** est une instruction spéciale, qui ne fait rien
- **assert**`(expr)` est une instruction qui vérifie que l'expression booléenne `expr` renvoie **True** ; si oui, ne fait rien, sinon interrompt le programme
- **import module** est une instruction Python permettant d'importer un ensemble de fonctionnalités d'un module externe (en particulier, bibliothèque standard)

Sortie

- Tout programme Python a une sortie standard : un endroit où le programme peut produire une sortie textuelle (un terminal, un fichier, etc.)
- Pour produire un contenu sur cette sortie standard, on peut utiliser la fonction prédéfinie `print` :
  `print(a, 42, "toto")` affiche, sur la sortie standard, la valeur de la variable `a`, l'entier 42 et la chaîne de caractères `toto`, séparés par des espaces et terminés par un retour à la ligne
- Utile pour afficher le résultat d'un calcul, vérifier au milieu d'un programme la valeur d'une variable, etc.; mais à n'utiliser que pour la sortie finale d'un programme, pas pour les résultats d'un calcul intermédiaire

Plan

Structures de contrôle

- Jusqu'ici on ne peut faire que des calculs séquentiels : pas de possibilité de changer le comportement d'un programme en fonction du résultat d'un calcul, ou de répéter certaines instructions
- Structure de contrôle : Instruction complexe permettant de changer le flot d'exécution d'un programme : en particulier, inclut les tests et les boucles
- En Python : principalement une instruction de test (**if**), deux instructions de boucle (**for**) et (**while**)

Test

```
# On évalue condition1 comme une expression booléenne
if condition1:
    # Si condition1 est vraie, ce code est exécuté
    ...
# Sinon on évalue la condition2
elif condition2:
    # Si elle est vraie, ce code est exécuté
    ...
else:
    # Si aucune des conditions n'est vraie, ce code est exécuté
    ...
```

- Il peut y avoir un nombre arbitraire de **elif** (y compris zéro)
- Le **else** est optionnel
- Les lignes **if**, **elif**, **else** se finissent par un « : »
- Indentation importante ! (4 espaces recommandés)

Boucle **for** : cas général

```
for x in collection:
    # On répète le code qui suit pour chaque élément
    # x de la collection (de type list, tuple, range,
    # set...). La variable x contient chaque élément
    # à la suite
    ...
```

- Au sein de la boucle **for**, on peut utiliser **continue** pour passer à l'élément suivant ou **break** pour interrompre la boucle avant la fin
- La ligne **for** finit par un « : »
- Indentation importante ! (4 espaces recommandés)

Boucle **for** : cas courant

```
for x in range(0, n):
    # On répète le code qui suit pour chaque entier
    # x entre 0 et n-1
    ...
```

- Au sein de la boucle **for**, on peut utiliser **continue** pour passer à l'élément suivant ou **break** pour interrompre la boucle avant la fin
- La ligne **for** finit par un « : »
- Indentation importante ! (4 espaces recommandés)

Boucle **while**

```
while condition:
    # On répète le code qui suit tant que
    # condition s'évalue à True
    ...
```

- Au sein de la boucle **while**, on peut utiliser **continue** pour passer à l'élément suivant ou **break** pour interrompre la boucle avant la fin
- La ligne **while** finit par un « : »
- Attention à ce que la condition devienne un jour vraie !
- Indentation importante ! (4 espaces recommandés)

Plan

Fonctions

- Tous les langages de programmation modernes (à l'inverse de certains des premiers langages de programmation) mettent l'accent sur la structuration du code au travers de fonctions (parfois appelées sous-routines, procédures, méthodes) correspondant à un ensemble d'instructions identifié par un nom et potentiellement utilisable dans différents contextes en l'appelant
- Une fonction a :

  Un nom : qui permet de lui faire référence

  Des arguments : une suite finie de noms de variables ; ces variables sont initialisées à chaque fois que la fonction est utilisée, à des valeurs choisies au moment de l'appel de la fonction

  Une valeur de retour : qui est retournée par la fonction à l'endroit où elle est appelée

Fonctions en Python

- Pas de typage des arguments ou de la valeur de retour d'une fonction, contrairement aux langages de programmation avec typage statique (mais une fonction s'attend souvent à avoir des arguments d'un type donné et renvoie souvent une valeur de retour donné)
- Toutes les variables créées à l'intérieur d'une fonction ne sont pas visibles à l'extérieur
- Bonnes pratiques pour rendre une fonction aussi réutilisable que possible :
  - Une fonction n'utilise pas de variables définies à l'extérieur, à l'exception des arguments
  - Une fonction ne réalise pas d'affichage ou d'autre effets de bords... sauf bien sûr si c'est le but de la fonction

Définition et appel de fonction

```
# On définit une fonction nom_fonction
def nom_fonction(arg1, arg2, arg3):
    # Au sein de la fonction, arg1, arg2 et arg3
    # ont les valeurs passées à la fonction
    ....
    # Après certains calculs, on renvoie une valeur
    return valeur


...

# Ailleurs dans le code
retour = nom_fonction(val1, val2, val3)
# retour contiendra la valeur de retour
# de la fonction appelée avec comme valeur
# des arguments val1, val2 et val3
```

Notes sur la définition de fonction

- Il peut y avoir zéro argument à une fonction
- Il peut ne pas y avoir de **return**, dans ce cas la fonction renvoie implicitement la valeur spéciale **None**
- Il peut y avoir plusieurs **return**, par exemple appelés dans des branches différents d'un test
- La ligne **def** finit par un « : »
- Indentation importante ! (4 espaces recommandés)
- La définition d'une fonction doit précéder le moment où la fonction est appelée

Fonctions récursives

- Une fonction peut tout à fait s'appeler elle-même : on dit que c'est une fonction récursive
- Souvent pratique pour certains calculs, par exemple :

```
def factorielle(n):
    if n<=1:
        return 1
    else:
        return n*factorielle(n-1)
```

Arguments et valeur de retour d'un programme

- Tout comme une fonction, un programme peut avoir des arguments : ils sont passés au programme quand celui-ci est exécuté (sur sa ligne de commande)
- On y a accès grâce à `sys.argv`, de type `list` (un **import sys**) est nécessaire
- Tout programme a aussi une valeur de retour ! Par convention, elle vaut 0 quand le programme s'est terminé correctement, une autre valeur sinon. On peut la renvoyer avec `sys.exit(valeur)`

Environnement d'exécution `__main__`

- Un fichier de code Python est parfois utilisé comme programme à part entière, parfois comme une bibliothèque de code annexe
- Pour distinguer entre ces deux cas, on peut utiliser la variable spéciale `__name__` : si celle-ci vaut `"__main__"`, on a appelé le code comme un programme
- On voit donc souvent la structure suivante :

```
import sys

# Définition de fonctions diverses, dont une
# fonction principale, souvent appelée main
def main(arguments):
    ...

# Teste si on a appelé le code comme un programme
if __name__ == "__main__":
    sys.exit(main(sys.argv))
```

Plan

Bibliographie – Algorithmique

- La plupart du cours (algorithmique) s'appuie sur ce livre : Cormen *et al.* (2009, 2010)
- Les chapitres correspondant (3ème édition) seront indiqués à l'occasion de chaque cours
- Le cours sera suffisant, pas la peine de lire le livre, mais peut être intéressant pour approfondir des parties
- Pour cette leçon, Chap. 1 uniquement

Bibliographie – Python

- Documentation en ligne de Python, en général très complète (mais pas toujours didactique) :
  `https://docs.python.org/fr/3/`
- Bon livre pour débuter en Python : Le Goff (2022)
- Livre très complet : Ramalho (2019)

Bibliographie I

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein : *Introduction to Algorithms*. MIT Press, 3e édition, 2009. ISBN 978-0262033848. URL http://mitpress.mit.edu/books/introduction-algorithms.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein : *Algorithmique*. Dunod, 3e édition, 2010. ISBN 978-2100545261. URL https://www.dunod.com/sciences-techniques/algorithmique-cours-avec-957-exercices-et-158-problemes.

Vincent Le Goff : *Apprenez à programmer en Python*. Eyrolles, 4e édition, 2022. ISBN 978-2416006555. URL https://www.eyrolles.com/Informatique/Livre/apprenez-a-nbsp-programmer-en-python-9782416006555/.

Bibliographie II

Luciano Ramalho : *Programmer en Python - Apprendre la programmation de façon claire, concise et efficace.* O'Reilly, 2019. ISBN 978-2412045145. URL https://www.oreilly.com/library/view/programmer-avec-python/9782412045145/.