PrépaBAC

Tle
GÉNÉRALE

NSI

Numérique & sciences informatiques

SPÉCIALITÉ

Cours & entraînement

NOUVEAU BAC

- Un cours complet et visuel
- Des projets guidés
- 150 exos & sujets de BAC
- Des corrigés expliqués

Pour réussir ton épreuve de spécialité !

INCLUS
Des mémos visuels

+

OFFERT Un abonnement au site de révision annabac

Hatier

PrépaBAC

Tle GÉNÉRALE

NSI

Numérique & sciences informatiques

SPÉCIALITÉ

Cours & entraînement

Guillaume Connan
Professeur agrégé de mathématiques
Enseignant de mathématiques et d'informatique
en classes préparatoires au lycée Externat-Chavagnes de Nantes
Formateur au DIU « Enseigner l'informatique au Lycée »

Vojislav Petrov
Professeur agrégé de mathématiques
Ancien élève de l'École normale supérieure de Lyon
Enseignant de mathématiques et d'informatique en classes
préparatoires au lycée Baimbridge en Guadeloupe

Gérard Rozsavolgyi
Professeur agrégé de mathématiques
Responsable de la Licence professionnelle métiers
de l'informatique à l'IUT d'Orléans
Formateur au DIU « Enseigner l'informatique au Lycée »

Laurent Signac
Docteur en informatique
Enseignant en informatique à l'école nationale supérieure
d'ingénieurs de Poitiers
Formateur au DIU « Enseigner l'informatique au Lycée »

ISBN : 978-2-401-09492-5

Hatier

Mode d'emploi

Voici les ressources que vous trouverez dans ce Prépabac :

Dans chaque chapitre

Un cours visuel

1. Des fiches synthétiques et illustrées
2. Des méthodes détaillées
3. Une carte mentale récapitulative

Un entraînement progressif

1. Des quiz pour se tester
2. Des exercices guidés
3. Des sujets de type bac

Des corrigés détaillés

Nous vous recommandons d'utiliser régulièrement l'ouvrage, en fonction de vos besoins. Ainsi vous pourrez aborder l'épreuve écrite et vos projets de NSI en toute sérénité, et acquérir les compétences nécessaires à la poursuite de vos études.

Les auteurs


Guillaume Connan


Vojislav Petrov


Gérard Rozsavolgyi


Laurent Signac

SOMMAIRE

Langages, programmation et algorithmique

1 Modularité et mise au point des programmes

COURS & MÉTHODES

1 Modules et documentations 10
2 Déboguer un programme 12
3 Tests 14
4 Guide de style 16
MÉMO VISUEL 18

EXERCICES & SUJETS SE TESTER • S'ENTRAÎNER 20
OBJECTIF BAC 25

CORRIGÉS 27

2 Récursivité

COURS & MÉTHODES

5 Fonctionnement d'un programme récursif 38
6 Méthode « diviser pour régner » 40
7 Exemples d'utilisation de la récursivité 42
MÉMO VISUEL 44

EXERCICES & SUJETS SE TESTER • S'ENTRAÎNER 46
OBJECTIF BAC 52

CORRIGÉS 56

3 Paradigmes de programmation

COURS & MÉTHODES

8 Les principaux paradigmes de programmation 66
9 Programmation fonctionnelle 68
10 Programmation orientée objet 70
MÉMO VISUEL 72

EXERCICES & SUJETS SE TESTER • S'ENTRAÎNER 74
OBJECTIF BAC 80

CORRIGÉS 84

Structures de données

4 Interface et implémentation des structures de données

COURS & MÉTHODES

11 Types abstraits 94
12 Implémentation des files 96
13 Structures de données Python 98
MÉMO VISUEL 100

EXERCICES & SUJETS SE TESTER • S'ENTRAÎNER . . 102
OBJECTIF BAC 108

CORRIGÉS 111

5 Structures arborescentes

COURS & MÉTHODES

14 Définition des arbres 122
15 Implémentation des arbres binaires 124
16 Utilisation des arbres 126
MÉMO VISUEL 128

EXERCICES & SUJETS SE TESTER • S'ENTRAÎNER 130
OBJECTIF BAC 134

CORRIGÉS 139

6 Graphes

COURS & MÉTHODES

17 Vocabulaire des graphes 152
18 Représentations d'un graphe 154
19 Parcours de graphes – Algorithme de Dijkstra . . 156
20 Applications de parcours de graphes 158
MÉMO VISUEL 160

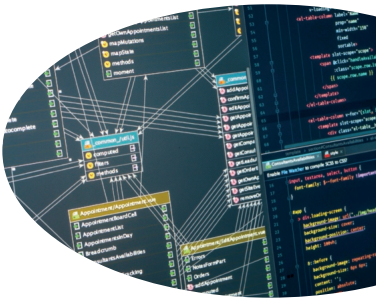
EXERCICES & SUJETS

SE TESTER • S'ENTRAÎNER 162
OBJECTIF BAC 167

CORRIGÉS 172

Bases de données

7 Conception de bases de données



COURS & MÉTHODES

21 Vocabulaire des bases de données 182
22 Le modèle relationnel 184
23 Conception de bases de données 186
MÉMO VISUEL 188

EXERCICES & SUJETS SE TESTER • S'ENTRAÎNER 190
OBJECTIF BAC 194

CORRIGÉS 197

8 Systèmes de gestion de bases de données – SQL

COURS & MÉTHODES

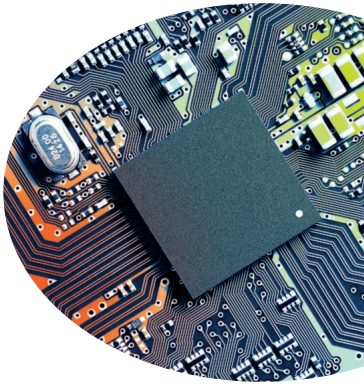
24 Les systèmes de gestion de bases de données . . . 206
25 Le langage SQL 208
26 Le langage SQL avancé 210
MÉMO VISUEL 213

EXERCICES & SUJETS SE TESTER • S'ENTRAÎNER 214
OBJECTIF BAC 220

CORRIGÉS 222

Architectures matérielles, systèmes d'exploitation et réseaux

9 Composants et processus



COURS & MÉTHODES

27 Les visages multiples du numérique 232
28 Processus et ressources 234
29 Notions d'interblocage 236
30 Composants intégrés d'un système sur puce . 238
MÉMO VISUEL 240

EXERCICES & SUJETS SE TESTER • S'ENTRAÎNER . . . 242
OBJECTIF BAC 248

CORRIGÉS 250

10 Protocoles de routage et sécurisation des communications

COURS & MÉTHODES

31 Le routage 260
32 Algorithmes de routage dynamique 262
33 La sécurisation des communications : HTTPS 264
MÉMO VISUEL 266
EXERCICES & SUJETS SE TESTER • S'ENTRAÎNER 268
OBJECTIF BAC 274
CORRIGÉS 276

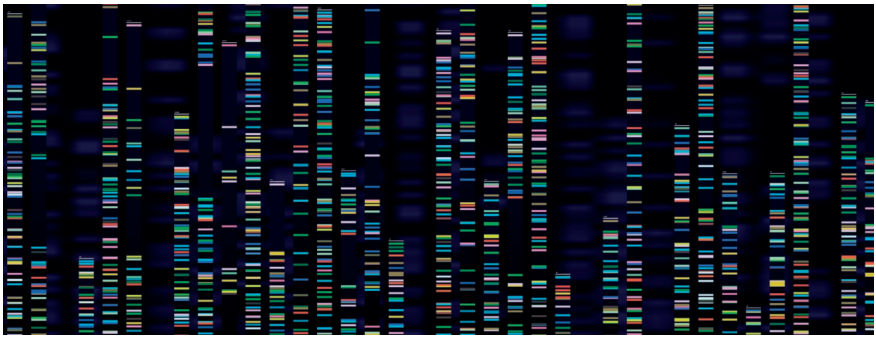
Programmation avancée et algorithmique

11 Programmes et données – Calculabilité

COURS & MÉTHODES

34 Programmes et données 290
35 Interprétation et compilation – Bytecode 292
36 Trois femmes entrées dans l'histoire de l'informatique 294
37 Histoire de la théorie de la calculabilité 296
38 Le problème de l'arrêt 298
MÉMO VISUEL 300
EXERCICES & SUJETS SE TESTER • S'ENTRAÎNER 302
OBJECTIF BAC 307
CORRIGÉS 310

12 Programmation avancée



COURS & MÉTHODES

39 Vers la programmation dynamique . . . 316
40 Programmation dynamique et alignement de séquences . . . 318
41 Recherche de motifs dans des chaînes de caractères . . . 320
MÉMO VISUEL . . . 322

EXERCICES & SUJETS SE TESTER • S'ENTRAÎNER . . . 324
OBJECTIF BAC . . . 329

CORRIGÉS . . . 332

Projets

13 Projets informatiques

EXEMPLES DE PROJETS

42 Reconstituer des arbres phylogénétiques . . . 342
43 Jouer à la pipopipette . . . 347

INDEX . . . 351

Crédits photographiques
3 h ph © kokouu/E+/Getty Images **3** m ph © Artur Tavares/EyeEm/Getty Images **3** b ph © MDart10/Shutterstock **4** h ph © Brooke Lark/Unsplash **4** m ph © Leamus/iStock/Getty Images Plus **4** b ph © pdm – stock.adobe.com **5** h ph © yurich84 – stock.adobe.com **5** m ph © canjoena – stock.adobe.com **5** b ph © Raimundas – stock.adobe.com **6** h ph © Sergey Nivens – stock.adobe.com **6** b ph © Ricky Deacon/iStock/Getty Images Plus **7** h ph © Tetiana Lazunova/iStock/Getty Images Plus **7** b ph © Westend61/Gustafsson/iStock/Getty-images **9** ph © kokouu/E+/Getty Images **37** ph © Artur Tavares/EyeEm/Getty Images **65** ph © MDart10/Shutterstock **93** ph © Brooke Lark/Unsplash **121** ph © Leamus/iStock/Getty Images Plus **151** ph © pdm – stock.adobe.com **181** ph © yurich84 – stock.adobe.com **205** ph © canjoena – stock.adobe.com **231** ph © Raimundas – stock.adobe.com **238** ph © akg-images **259** ph © Sergey Nivens – stock.adobe.com **289** ph © Ricky Deacon/iStock/Getty Images Plus **294** h Coll. British Government art collection **294** b Coll. Shorpy Photo Archives **295** Coll. Nasa **296** h ph © DR **296** b Coll. wikipedia **297** h Coll. wikipedia **297** b ph © DR **315** ph © Tetiana Lazunova/iStock/Getty Images Plus **341** ph © Westend61/Gustafsson/iStock/Getty-images **342** Coll. Archives Hatier

1 Modularité et mise au point des programmes

Toyota rappelle 600 000 voitures après un bug informatique.

L'écriture du code ne représente pas la part la plus importante du développement informatique. Le débogage, puis la maintenance sont bien plus longs qu'on ne pourrait l'imaginer.

FICHES DE COURS

1 Modules et documentations 10
2 Déboguer un programme 12
3 Tests 14
4 Guide de style 16
MÉMO VISUEL 18

EXERCICES & SUJETS

SE TESTER Exercices 1 à 6 20
S'ENTRAÎNER Exercices 7 à 11 22
OBJECTIF BAC Exercice 12 • Sujet guidé 25

CORRIGÉS

Exercices 1 à 12 27

1 Modules et documentations

En bref *Que l'on distribue du code ou que l'on utilise l'offre pléthorique de modules Python, la manipulation des modules et de la documentation fait partie intégrante du développement logiciel.*

I Programmation modulaire

1 Principes généraux

■ Le **principe de modularité** est particulièrement important dans tout développement logiciel. Il simplifie les tests, permet de réutiliser du code, et facilite la maintenance. Ce principe peut opérer à plusieurs niveaux :
– découper le code en fonctions ;
– grouper les fonctions dans une classe (on les appelle alors méthodes) ;
– grouper des fonctions par thèmes, dans un fichier séparé ;
– grouper des fichiers en bibliothèques.

■ La **programmation modulaire** intervient :
– lors de l'écriture de portions de code pour les réutiliser plus tard, ou les distribuer ;
– lors de l'utilisation, pour répondre à un besoin, de code écrit par un·e autre.

2 Modularité en Python

■ On peut (on doit...) écrire des **fonctions** et des **procédures**, et les réutiliser.

■ Python permet de faire de la **programmation orientée objets** → FICHE 10 et donc de grouper des fonctions (alors appelées méthodes) au sein de classes, selon le type d'objet sur lequel elles agissent.

■ Fonctions, procédures et classes peuvent être groupées par thèmes dans des fichiers séparés alors appelé **modules** qui peuvent être groupés en **packages**.

II Importation des modules

■ Certains modules Python sont installés par défaut (bibliothèque standard) et d'autres peuvent être ajoutés en utilisant des outils comme `pip`. Le dépôt Pypi (Python Package Index, https://pypi.org/) référence la plupart des modules tiers.

■ Pour importer un module installé, on utilise le mot clé `import` dans une de ses variantes. Voici des exemples avec le module `random` :

• On importe le module `random`, les fonctions du module doivent être préfixées du nom du module :

```
import random
a = random.randint(1, 10)
```

• On importe le module `random` et on le renomme en `rnd` (le nouveau nom donné est généralement plus court) :

```
import random as rnd
a = rnd.randint(1, 10)
```

• On importe uniquement la fonction `randint` du module `random` (plus de préfixe, mais seule la fonction `randint` est disponible) :

```
from random import randint
a = randint(1, 10)
```

■ Toutes les variantes contenant des * dans la ligne d'importation visent à rendre disponibles des fonctions *sans avoir à les préfixer*, ce qui peut paraître efficace pour le programmeur débutant. En réalité, il n'en est rien : on ne maîtrise plus la liste exacte de ce qui est importé dans l'espace de noms principal, ce qui peut être source de bugs complexes à découvrir. De plus, en reprenant plus tard son travail ou en le distribuant, on n'aura plus de trace simple de l'endroit où se trouvent les fonctions et classes utilisées, rendant la maintenance difficile : `import *` est à bannir !

III Documentation

■ Que l'on écrive ou qu'on utilise une fonction ou un module, la documentation est centrale pour que le travail soit réutilisable.

Parmi les différents mécanismes, l'un des plus simples est la **docstring** qui peut être attachée à une fonction, à une procédure, à une méthode, à une classe, à un module, ou à un package. Dans tous les cas, c'est une chaîne de caractères qui doit figurer au début de l'entité qu'elle documente.

Cette documentation est ensuite consultable à l'aide de la commande `help` :

```
>>> import random
>>> help(random) # affiche la docstring du module
>>> help(print)  # affiche la docstring de la fonction
```

Exemple d'écriture d'une docstring :

```
def factorielle(n) :
    """ Calcul de la factorielle :
        n! = 1x2x…xn
        >>> factorielle(5)
        120
    """
    res = 1
    for i in range(2, n + 1):
        res = res * i
    return res
```

2 Déboguer un programme

En bref *La phase de débogage d'un programme, qui consiste à rechercher les erreurs de programmation, est très gourmande en temps. Ceci est particulièrement vrai avec Python dont le typage dynamique repousse la découverte des fautes au moment de l'exécution.*

I Le traceback de Python

- Lorsque l'interpréteur Python rencontre un problème, une **exception** est levée. Si elle n'est pas *capturée*, cette exception provoque l'arrêt du programme et l'affichage d'un message appelé **traceback**. Ce message permet de connaître la nature et le contexte de l'incident.
- Lire l'intégralité des messages d'erreur et se familiariser avec eux permet à la longue de gagner énormément de temps.

II Erreurs de syntaxe

Les erreurs de syntaxe empêchent l'interpréteur de comprendre le code écrit et provoquent la levée d'une exception **avant même l'exécution du code**. Ces erreurs sont souvent faciles à trouver :
– parenthèse, crochet ou guillemet mal fermé (`SyntaxError`) ;
– mauvaise indentation (`IndentationError`).

À NOTER
Attention, le **traceback** indique la ligne où l'erreur a été détectée (qui n'est pas forcément la ligne où l'erreur a été commise).

Exemple :

```
1 for i in range(1, 10):
2     print("Quel est le carré de {} ?".format(i)
3     print("C'est : {}".format(i**2))
```

```
  File "tst.py", line 3
    print("C'est : {}".format(i**2))
        ^
SyntaxError : invalid syntax
```

L'erreur, signalée ligne 3, est en fait ligne 2 : **il manque une parenthèse fermante.** Comme les instructions peuvent courir sur plusieurs lignes, elle n'est détectée que ligne 3 !

À NOTER
Mélanger des **espaces** et des **tabulations** pour indenter le code est souvent incorrect, et toujours déconseillé. Ce type d'erreur est difficile à repérer car la largeur d'une tabulation peut correspondre à celle de quelques espaces. Il est conseillé de configurer son éditeur pour que l'appui sur la touche tab provoque l'insertion de 4 espaces (qui est le niveau d'indentation conseillé en Python).

III Erreurs à l'exécution

■ Les exceptions levées à l'exécution sont plus difficiles à trouver, car elles nécessitent de comprendre (à différents degrés) l'exécution du code. Elles sont aussi plus variées :

- `NameError` : un nom de variable a-t-il été mal orthographié ?
- `IndexError` : l'indice utilisé est-il en dehors de la liste ?
- `TypeError` : a-t-on essayé d'ajouter un nombre à une chaîne de caractères ?

■ Outre le type d'exception et la ligne l'ayant levée, le traceback contient un historique des appels de fonctions (la pile des appels) permettant de connaître le contexte d'exécution. La recherche d'une erreur s'apparente alors à une enquête : depuis l'endroit où l'erreur s'est déclarée, on remonte le fil d'exécution pour en déterminer la cause, qui peut être à un tout autre endroit (le traceback se lit pour cette raison de bas en haut) :

```
Traceback (most recent call last) :
  File "error.py", line 10, in <module>
    f2()
  File "error.py", line 7, in f2
    f1()
  File "error.py", line 3, in f1
    a = a / (b + c)
ZeroDivisionError : division by zero
```

■ Lors de l'exécution du fichier `error.py`, une exception de type `ZeroDivisionError` a été levée, ligne 3 (`a = a / (b + c)`), dans la fonction `f1`. La fonction en question a été exécutée suite à un appel sur la ligne 7, dans la fonction `f2`, qui elle-même a été appelée par la ligne 10 du programme principal.

À NOTER

Grace Hopper → PRÉPABAC NSI 1re, FICHE 25 rapporte en 1947 un cas de bug devenu célèbre : il s'agissait d'un insecte (*bug* en anglais) ayant provoqué des erreurs de calcul dans un ordinateur Mark II.

IV Débogueur

■ Le débogueur permet de dérouler un programme pas à pas et de vérifier l'état de chaque variable. C'est un outil parfois indispensable pour comprendre des bugs complexes.

■ Le débogueur post-mortem permet d'inspecter l'état du programme (contenu des variables par exemple) à partir du lieu où l'exception a été levée. C'est souvent suffisant pour comprendre une erreur.

La plupart des environnements de développement proposent un débogueur.

3 Tests

En bref *Plutôt que de prouver la justesse d'un programme, on se contente souvent de tester le code en vérifiant qu'il se comporte correctement sur des entrées pour lesquelles le résultat est connu.*

I Tests unitaires

■ De l'écriture des spécifications à la validation d'un logiciel, chaque étape du développement est idéalement accompagnée d'une phase de tests. Nous nous intéressons ici aux **tests unitaires** dont l'objectif est de tester indépendamment chaque fonction.

■ Ces tests peuvent être conçus avant ou après la fonction à tester, éventuellement par une personne différente. Ils sont généralement exécutés aussi souvent que possible pour éviter les problèmes de **régression**.

MOT CLÉ
Régression : ajout de bugs lors de modification du code.

■ Les tests unitaires doivent être les plus couvrants possible, c'est-à-dire envisager le plus de cas possibles (simples ou à problème). Chaque branche du code doit idéalement être testée.

II Outils de test

■ Il existe plusieurs outils permettant de réaliser des tests unitaires : les assertions, les doctests, le module tiers `pytest`...

■ Les exemples suivants concernent une fonction `fibo(n)` qui calcule le terme n de la suite de Fibonacci (suite définie par $F_0 = 0$, $F_1 = 1$ et, pour $n > 1$, $F_n = F_{n-1} + F_{n-2}$).

1 Assertions

Une assertion échoue si l'expression booléenne qui suit le mot clé `assert` est fausse. Si elle est vraie, l'exécution continue sans erreur. Une assertion permet donc de vérifier, par exemple, le retour d'une fonction. Dès qu'une assertion est fausse, l'exécution s'arrête (exception `AssertionError`).

Exemple :

```
assert fibo(0) == 0
assert fibo(3) == 2
assert fibo(7) == 13
```

2 Module `doctest`

■ Le module `doctest` permet d'intégrer des tests dans la *docstring* des fonctions → FICHE 1. Les doctests sont repérées par la chaîne `>>>`. Écrire des doctests permet à la fois de réaliser des tests unitaires, mais aussi de documenter efficacement la fonction.

Exemple :

```
# Fichier fibo.py
def fibo(n: int) -> int:
    """
    fibo(n) : n-ième terme de la suite de Fibonacci
    >>> fibo(3)
    2
    >>> fibo(7)
    13
    """
    # Texte de la fonction
```

- Les tests peuvent être validés depuis l'interpréteur Python (ici, un des tests a échoué) :

```
>>> import fibo
>>> import doctest
>>> doctest.testmod(fibo)
Failed example:
    fibo(7)
Expected: 13
Got: 15
```

3 Module pytest

- `pytest` permet de faire des tests plus complets (vérification qu'une exception est levée par exemple) et propose des diagnostics plus complets et plus explicites que les autres outils.
- On peut démarrer l'utilisation de `pytest` par l'écriture de fonctions préfixées par `test_` contenant chacune une assertion.

Exemple :

```
# Fichier test_fibo.py
import pytest
from fibo import fibo
def test_0():
    assert fibo(0) == 0
def test_3():
    assert fibo(3) == 2
def test_7():
    assert fibo(7) == 13
def test_neg():
    with pytest.raises(ValueError):
        fibo(-1)
```

- L'exécution des tests réalisés avec `pytest` montre un diagnostic plus précis des problèmes (non reproduit ici).

```
>>> import pytest
>>> pytest.main(["test_fibo.py"])
```

4 Guide de style

En bref *On oublie souvent qu'un programme sera relu et modifié par un humain : son auteur ou un autre développeur. Adopter un style d'écriture standard facilite cette relecture.*

I Choisir correctement des noms

- Syntaxiquement, les noms de variables, de fonctions, de classes, de méthodes, d'attributs → FICHE 10 peuvent comporter des lettres, des chiffres, des caractères « _ » et ne doivent pas commencer par un chiffre.
- Quel que soit le langage, on choisira un nom d'autant plus **évocateur** qu'il est utilisé dans une grande portion de code.
- La PEP 8 (guide de style Python accessible sur https://www.python.org/dev/peps/peps-0008/) ajoute les conventions suivantes :

• Noms de modules courts, en minuscules et de préférence sans « _ ».
Exemple : `crypto`.

• Noms de classes ou de types en *CamelCase*, c'est-à-dire en minuscules, sans « _ », avec une majuscule au début de chaque mot.
Exemples : `class NombrePremier`.

• Noms de fonctions, de méthodes, d'attributs ou de variables en minuscules, les mots séparés par des « _ ».
Exemple : `def decomposition_facteurs_premiers(...)`.

• Les variables utilisées comme constantes sont en majuscules, les mots séparés par des « _ ».
Exemple : `TOTAL_MAX = 10`.

II Espaces, indentations, lignes blanches

1 Indentation

- Les blocs Python sont délimités par l'indentation. La PEP 8 propose d'indenter les blocs à l'aide de 4 espaces (la plupart des éditeurs Python utilisent ce réglage).
- Une ligne ne devrait pas excéder 79 caractères (cette règle est parfois transgressée). Lorsqu'une instruction court sur plusieurs lignes, on facilite la lecture en indentant. Exemple tiré de la PEP 8 :

```
# Aligner avec la parenthèse ouvrante
foo = nom_de_fonction_long(var_one, var_two,
                           var_three, var_four)
```

- Enfin, on écrit généralement une seule instruction par ligne.