Cours de Progammation en Python

Gradi KAMINGU LUBWELE, M.Sc.

Laboratoire d'Analyse–Recherche en Économie Quantitative (LAREQ)

11 février 2019

Table des matières

AVANT-PROPOS

Aujourd'hui, il existe de nombreux ouvrages abordant le sujet de la programmation en général et la programmation en Python en particulier. Nous pouvons citer particulièrement le livre de Gerard Swinnen intitulé *Apprendre à programmer avec Python*, celui de Mark Lutz et David Ascher intitulé *Python en concentré* ou encore *Programmation Python. Conception et Optimisation* de Tarek Ziadé. Ces ouvrages sont très bien rédigés et sont plus détaillés que le présent guide. Nous conseillons donc aux lecteurs qui veulent avoir des amples informations et veulent apprendre d'autres techniques avancées de la programmation en Python, de bien vouloir les lire.

Cependant, ces notes de cours sont élaborées, initialement, pour les étudiants de deuxième licence de l'Académie de Science et de Technologie, mais elles peuvent servir également aux étudiants (en sciences de gestion, en sciences économiques, en sciences mathématiques, en sciences de ingénieur, etc.), quelque soit la filière, dont la formation nécessite la programmation scientifique. Les étudiants ou développeurs débutants ne connaissant pas encore ce langage, trouverons ces notes comme un guide servant comme tremplin dans le langage. Ainsi, pour une lecture plus aisée, le lecteur devra avoir des connaissances de base en algorithmique et logique de programmation.

PLAN

Nous ne pouvons pas clore le présent avant-propos sans remercier les gens qui ont participé d'une manière ou d'une autre à l'élaboration de ces notes de cous. Nous pensons particulièrement au professeur Pierre Kafunda, qui nous a donné l'opportunité de rédiger ce document et à l'assistant Yves Mbolo, qui nous a assisté dans l'élaboration, tout comme dans la préparation des travaux pratiques du cours. Nous remercions également étudiants de deuxième licence de l'Académie de Science et de Technologie, de l'année académique 2018 – 2019, notamment Acacia Nday, Benie Madiata, Daniel Zema et Percide Lumbu.

Gradi L. Kamingu

Chapitre 1

PRÉLIMINAIRES SUR LE LANGAGE PYTHON

1.1 Rappels sur la programmation

1.1.1 Définition (Programme informatique)

*Un **programme informatique** est un ensemble d'opérations destinées à être exécutées par un ordinateur pour résoudre un problème donnée.*

Pour écrire un programme, on utilise un *langage de programmation.*

1.1.2 Définition (Langage de programmation)

*On appelle **langage de programmation** est un système de symboles utilise pour la description des procédés de résolution de problèmes au moyen de l'ordinateur.*

Similairement avec le langage humain (ou langue naturelle), le langage de programmation possède :

— un alphabet ;
— un vocabulaire ;
— de règles de grammaire (la syntaxe) ;
— de significations (la sémantique).

1.1.3 Types de langages de programmation

Les langages de programmation peuvent être classés en deux groupes qui sont : *(i)* les langages de bas niveau et *(ii)* les langages de haut niveau.

A. Langages de bas niveau

*Un **langage de bas niveau** est un langage de programmation qui permet d'écrire des programmes en tenant compte des caractéristiques particulières de l'ordinateur censé exécuter le programme.*

Ce sont des langages permettant de manipuler explicitement des registres, des adresses mémoires, des instructions machines.

Ainsi, on distingue deux type des langages de bas niveau : le langage machine et le langage d'assemblage.

A.1. Langage machine

Le langage machine est la suite des bits (des « 0 » et des « 1 ») interprétable directement par le processeur d'un ordinateur.
C'est le langage natif du processeur. Ce langage est composée d'instructions et des données à traiter sous forme binaire.

A.2. Langage d'assemblage

Le langage d'assemblage ou langage assembleur est un langage de bas niveau qui représente les combinaisons des bits du langage machine par des codes mnémoniques, facile à retenir ou lisible à l'homme. Ce langage est composée d'instructions et des données à traiter sous forme binaire.

B. Langages de haut niveau

*Un **langage de haut niveau** ou **langage évolué** est un langage de programmation qui permet d'écrire des programmes en utilisant des mots usuels des langues naturelles (très souvent de l'anglais) et des symboles mathématiques usuels.*

Ce sont des langages proches des langues naturelles. Le premier langage de haut niveau est le Fortran (en 1954), suivi ensuite du Lisp (en 1958), de l'Algol (en 1958) et du COBOL (en 1959).

En tenant compte du paradigme ou approche de programmation, on distingue plusieurs types de langages de programmation de haut niveau.

B.1. Langages de programmation impérative

*Un **langage de programmation impérative** est un langage de programmation qui décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme.*

B.2. Langages de programmation déclaratives

*Un **langage de programmation déclarative** est un langage de haut niveau qui consiste à déclarer les données du problème et demander au programme de le résoudre.*

B.2.1. Langages de programmation fonctionnelle

*Un **langage de programmation fonctionnelle** est un langage de programmation déclarative où les actions reposent sur les fonctions mathématiques.*

Exemples 1.1.1. *Lisp, Ruby, Scarla.*

B.2.2. Langages de programmation logique

*Un **langage de programmation fonctionnelle** est un langage de programmation déclarative qui consiste à exprimer les problèmes et les actions sous forme de prédicats (en utilisant la logique du premier ordre).*

Exemple 1.1.1. *Prolog.*

B.2.3. Langages de programmation descriptive

*Un **langage de programmation descriptive** est un langage de programmation déclarative qui permet de décrire des structures des données. Ce type de langage est spécialisé dans l'enrichissement d'information textuelle.*
Il utilise des *balises*, qui sont des unités syntaxiques servant à délimiter une séquence de caractères ou marquant une position précise à l'intérieur d'un flux de caractères.

Exemples 1.1.2. *LaTeX, HTML, XML.*

B.3. Langage de programmation orientée-objet

*Un **langage de programmation orientée-objet** ou un **langage de programmation par objet** est un langage de haut niveau consistant en la définition et l'assemblage de composantes logicielles appelées objets. Une fédération d'objets forme une **classe**.*

Exemples 1.1.3. *C#, C++, Java, VB.net, Effeil.*

1.1.4 Traducteur

Un programme écrit dans un langage de programmation non natif au processeur peut être exécuté sur plusieurs machines moyennant des petites modifications. Ainsi, les programmes écrits en langage non natif au processeur sont d'abord traduits en suite d'instructions machines avant exécution. Ce programme spécial est appelé **traducteur**.

Parmis les traducteurs, on distingue *les assembleurs*, *les compilateurs* et *les interpréteurs*.

A. Assembleur

*Un **assembleur** est un programme qui permet de convertir des codes mnémoniques du langage d'assemblage en langage machine.*

B. Interpréteur

*Un **interpréteur** est un programme qui traduit ligne par ligne dès la saisie du programme.*

Un langage de programmation est dit *interprété* lorsqu'il ses opérations d'analyses et de traductions sont effectuées par un interpréteur.

Exemples 1.1.4. *BASIC, MATLAB, Perl, Prolog, PHP.*

C. Compilateur

*Un **compilateur** est un programme qui transforme l'intégralité de programme écrit en un langage de haut niveau (code source) en langage machine.*

Un langage de programmation est dit *compilé* lorsqu'il ses opérations d'analyses et de traductions sont effectuées par un compilateur.

Exemples 1.1.5. *LaTeX, C#, C, C++, Fortran, Cobol.*

1.2 Présentation de l'environnement Python

Python est un langage de programmation de script de haut niveau portable et gratuit, doté d'un typage dynamique fort, d'une gestion automatique de la mémoire et d'un système de gestion

des exceptions. Python est développé depuis 1989 par **Guido van Rossum** et de nombreux contributeurs bénévoles. Sa première version fut publiée le 20 fevrier 1991 et sa derniere version date du 1er mai 2018.

Python est un langage de programmation **interprété**, c'est-à-dire que les instructions qu'on lui envoie sont ≪traduites≫ en langage machine au fur et à mesure de leur lecture. Cette logique est différente des langages comme le C/C++, qui sont des **langages compilés** car, avant de pouvoir les exécuter, un logiciel spécialisé se charge de transformer tout le code du programme en langage machine.

1.3 Caractéristiques du Langage Python

Python est un langage de programmation qui est choisi pour plusieurs raisons. Les caractéristiques que possèdent le langage fait que plusieurs utilisateurs (développeurs d'applications, économistes, data scientists, statisticiens, experts en modélisation, ingénieurs, etc.) s'y intéressent. Parmi les caractéristiques qui ont séduit les utilisateurs de Python, nous pouvons citer les suivantes :

- Python est **gratuit**, c'est-à-dire qu'on on peut l'utiliser sans restriction dans des projets commerciaux.
- Python est **multi-paradigme**, car il favorise la programmation **orientée-objet** (tous les types de base, les fonctions, les instances des classes et les classes elles-mêmes sont des objets), **impérative structurée**, **procédurale** (il applique la méthodologie par décompositions successives en utilisant les procédures et les fonctions) et **fonctionnelle** (il dispose des compréhensions des listes, des dictionnaires et des ensembles).
- Python est **multi plate-forme**, car il est disponible sur les plates-formes comme des smartphones et les ordinateurs centraux. De plus, il est disponible sur plusieurs systèmes d'exploitation telles que les variantes d'Unix (GNU/Linux, Debian, Android, etc.), mais aussi sur plusieurs systèmes d'exploitation propriétaires tels que iOS, MacOS, BeOS, NeXTStep, MS-DOS, NetBSD, FreeBSD et les différentes versions de Windows.
- Python est (optionnellement) **multi-threadé**.
- Python est **dynamique** (l'interpréteur peut évaluer des chaînes de caractères représentant des expressions ou des instructions Python), **orthogonal** (un petit nombre de concepts suffit à engendrer des constructions très riches), **réflectif** (il supporte la métaprogrammation, par exemple la capacité pour un objet de se rajouter ou de s'enlever des attributs ou des méthodes, ou même de changer de classe en cours d'exécution) et **introspectif** (un grand nombre d'outils de développement, comme le debugger ou le profiler, sont implantés en Python lui-même).
- Python est **extensible**, c'est-à-dire qu'il permet l'interfaçage facile avec des bibliothèques C existantes. Aussi, il peut interagir avec d'autres logiciels de traitement des données, des systèmes de gestion de base des données et des systèmes d'information géographiques.
- Python est **en évolution continuelle**, car il soutenu par une communauté d'utilisateurs enthousiastes et responsables, dont la plupart sont des supporters du logiciel libre. Parallèlement à l'interpréteur principal, écrit en C et maintenu par le créateur du langage, un deuxième interpréteur, écrit en Java, est en cours de développement.

1.4 Environnements de Développement Intégré pour le langage Python

*Un **Environnement de Développement Intégré** ou en anglais **Integrated Development Environment (IDE)** est une plateforme permettant d'éditer et exécuter les codes écrits dans un langage de programmation donné.*

1.4.1 Jupyter Notebook

Jupyter Notebook (formellement connu sous le nom de **The IPython Notebook**) est un IDE interactif dans lequel on peut combiner l'exécution des codes, le texte riche, le multimédia et les équations mathématiques et les graphiques. À l'aide de cet Ide, le programmeur peut créer et partager les documents. Il est supporté par plusieurs systèmes d'exploitation, notamment le Linux, Windows et MacOS.

Figure 1.1 – Logo de Jupyter Notebook.

1.4.2 Anaconda

Anaconda est une distribution libre pour les langages de programmation Python et R, dans les applications scientifiques. Dans cette distribution, on peut avoir accès à plusieurs autres applications qui peuvent être des IDEs, des bibliothèques, etc. Par défaut, on a les applications suivantes :

(1) JupyterLab ;
(2) Jupyter Noteboook ;
(3) QtControl ;
(4) Spyder ;
(5) Glueviz ;
(6) Orange ;
(7) Rstudio ;
(8) Visual Studio Code.

Figure 1.2 – Logo de Anaconda.

1.5 Variables

1.5.1 Définition (Variable)

*Une **variable** est une zone de la mémoire dans laquelle on stock une valeur pouvant changer au cours de l'exécution du programme.*

1.5.2 Déclaration d'une variable

Pour pourvoir utiliser une variable, il faut le déclarer. En Python, la déclaration d'une variable et son initialisation (c'est-à-dire la première valeur que l'on va stocker dedans) se font en même temps. Pour faire la déclaration avec initialisation, on utilise le symbole d'*affection* « = ».
Une ***affectation*** *est une action d'attribuer ou assigner une valeur à un type de données quelconque.*

Exemple 1.5.1.

```
In [1]: x=4

In [2]: x
Out[2]: 4
```

Dans l'exemple 1.5.1, nous avons déclaré, puis initialisé la variable x avec la valeur 4. Cependant, trois choses se sont passées dans l'exécution des ces lignes des codes :

— Python a *compris* que la variable était un entier. On dit que Python est un langage au *typage dynamique*.
— Python a réservé l'espace en mémoire pour y accueillir un entier (chaque type de variable prend plus ou moins d'espace en mémoire), et a fait en sorte qu'on puisse retrouver la variable sous le nom x.
— Python a affecté la valeur 4 à la variable x.

1.5.3 Types des variables

Le **type** d'une variable correspond à la nature de celle-ci. Les variables peuvent être des numériques ou des chaînes des caractères (*string* ou *str*).
Pour les numériques, on distingue les entiers (*integer* ou *int*), les reels (*float*) et les complexes (*complex*).

Exemple 1.5.2.

```
In [1]: a=4

In [2]: b=4.0

In [4]: c=4j

In [5]: e=complex(4.0, 10)

In [6]: d="Je suis Kamingu!"

In [7]: f='Je suis Percide!'
```

On remarque que Python reconnaît certains types de variable automatiquement (entier, réel). Par contre, pour une chaîne de caractères, il faut l'entourer de guillemets (simples, doubles voire trois guillemets successifs simples ou doubles) afin d'indiquer à Python le début et la fin de la chaîne.

Pour reconnaître ce type, il suffit d'utliser la syntaxe suivant :

type(`nom_de_la_varaible`)

Exemple 1.5.3.

```
In [1]: type(a)
Out[1]: int

In [2]: type(b)
Out[2]: float

In [3]: type(c)
Out[3]: complex

In [4]: type(e)
Out[4]: complex

In [5]: type(d)
Out[5]: str

In [6]: type(f)
Out[6]: str
```

Nota 1.5.1. *Python propose un moyen simple de permuter deux variables (échanger leur valeur). Dans d'autres langages, il est nécessaire de passer par une troisième variable (comme dans l'exemple 1.5.4) qui retient l'une des deux valeurs. Mais on peut aussi utiliser la structure de l'exemple 1.5.5*

Exemple 1.5.4.

```
In [1]: var1=100

In [2]: var2=400

In [3]: var3=var2

In [4]: var2=var1

In [5]: var1=var3

In [6]: var1
Out[6]: 400

In [7]: var2
Out[7]: 100
```

Exemple 1.5.5.

```
In [1]: var1=100

In [2]: var2=400

In [3]: var1, var2 = var2, var1

In [4]: var1
Out[4]: 400

In [5]: var2
Out[5]: 100
```

1.5.4 Affectation multiple

En Python, on peut aussi affecter assez simplement une même valeur à plusieurs variables.

Exemple 1.5.6.

```
In [1]: a = b = c = d = 4.0

In [2]: c
Out[2]: 4.0
```

On peut aussi effectuer des affectations parallèles à l'aide d'un seul opérateur.

Exemple 1.5.7.

```
In [1]: a, b, c = 2, 4.0, "Je suis Gradi"

In [2]: a
Out[2]: 2

In [3]: b
Out[3]: 4.0

In [4]: c
Out[4]: 'Je suis Gradi'
```

1.5.5 Convention d'Écriture

— Par convention, les noms des variables se feront par des lettres minuscules `a-z` et majuscule `A-Z`, les chiffres du système decimal `0-9`, le point « . » et le caractère de soulignement « _ ».
— Un nom de variable ne peut pas commencer par un chiffre. De plus, s'il commence par un point, le deuxième caractère ne peut pas être un chiffre.
— Python est sensible à la casse. Il distingue les majuscules et les minuscules. En effet, `ASCITECH`, `AsciTech`, `Ascitech` et `ascitech` sont des variables différents.
— Pour un objet ayant un nom composé, il est conseillé d'utiliser les majuscules pour le(s) mot(s) intérieurs et le mot final. On utilisera par exemple, `revenuParHab` pour exprimer le revenu par habitant et `tauxDeChange` pour exprimer le taux de chance.
— Certains mots appartiennent à Python et sont utilisés comme mots-clés (voir le tableau suivant).

and	del	from	none	true
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	false	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

Table 1.1 – Liste des mots-clés de Python

1.5.6 Transtypage

On peut passer d'un type d'objet à un autre. Dans ce moment, on doit utiliser une technique qu'on appel *transtypage*, en anglais *casting*.

En Python, pour effectuer le transtypage, la syntaxe appropriée est `nouveau_type(nom_variable)`.

En Python, le transtypage se fait avec les fonctions `int()`, `float()` et `str()`.

Exemple 1.5.8.

```
In[1]: x=4

In[2]: y=4.5

In[3]: z="521"

In[4]: str(y)
Out[4]: '4.5'

In[5]: int(y)
Out[5]: 4

In[6]: float(x)
Out[6]: 4.0
```

1.5.7 Opérations en Python

A chaque type de variable correspond différentes opérations. Nous allons ainsi citer les opérations que nous allons utiliser tout au long de notre cours.

- Pour les données de type numérique, les opérations utilisées sont : + (addition), − (soustraction), ∗ (multiplication), / (division décimale), ∗∗ (puissance), // (modulo ou reste de la division entière), % (division entière).
- Pour les chaînes des caractères, les opérations utilisées sont : + (concaténation ou jonction des chaînes de caractères) et ∗ (duplication d'une chaîne).

Exemple 1.5.9.

```
In  [1]: x = 1900

In [2]: y = 4

In [3]: z = "Pierre"

In [4]: res1 = x - 4*y

In [5]: res1
Out[5]: 1884

In [6]: res2 = x**y

In [7]: res2
Out[7]: 13032100000000
```

```
In [8]: res3 = x/y

In [9]: res3
Out[9]: 475.0

In [10]: res4 = x//y

In [11]: res4
Out[11]: 475

In [12]: res5 = x%y

In [13]: res5
Out[13]: 0

In [14]: res6 = z*3

In [15]: res6
Out[15]: 'PierrePierrePierre'
```

Nota 1.5.2. *Une opération illicite est toujours signalée par Python au moyen d'un message d'erreur.*

```
In [16]: res6 + res3
Traceback (most recent call last):

File "<ipython-input-60-331547e75c07>", line 1, in <module>
res6 + res3

TypeError: Cannot convert 'float' object to str implicitly
```

1.6 Écriture à l'écran

1.6.1 Affichage simple

En Python, la fonction dédiée pour faire l'affichage est **`prinf()`**.

Étudions cette fonction avec l'exemple suivant :

Exemple 1.6.1.

```
In [1]: age = 18

In [2]: nom = "Gradi"

In [3]: print(nom, "a", age, "ans")
Gradi a 18 ans
```

On remarque que quand Python interprète cet appel de fonction, il va afficher les paramètres dans l'ordre de passage, en les séparant par un espace.

Pour afficher deux chaînes de caractères l'une à côté de l'autre sans espace, on utilise l'opération de concaténation, comme dans l'exemple suivant :

Exemple 1.6.2.

```
In [1]: nom = "Acacia"

In [2]: fonction = "etudiante"

In [3]: print(nom+fonction)
Acaciaetudiante
```

1.6.2 Affichage avec écriture formatée

Pour afficher avec un meilleur format, Python met à la disposition des programmeurs la méthode `format`.

A. Formatage avec les accolades

(1) Pour inserer le contenu de la variable lors de l'affichage, on utilise les accolades vides {}.

Exemple 1.6.3.

```
In [1]: age = 32

In [2]: prenom = "Tresor"

In [3]: print("{} a {} ans".format(prenom, age))
Tresor a 32 ans
```

(2) Python nous donne aussi la possibilité d'afficher (avec la méthode `format`) en indiquant l'ordre d'apparition lors de l'affichage. À savoir que Python commence toujours le comptage à 0.

Exemple 1.6.4.

```
In  [1]: nom1 = "Pupa"

In  [2]: nom2 = "Coen"

In  [3]: print("{0} et {1} sont amis".format(nom1, nom2))
Pupa et Coen sont amis

In  [4]: print("{1} et {0} sont amis".format(nom1, nom2))
Coen et Pupa sont amis
```

B. Formatage avec les accolades et caractères de formatage

Python nous donne la possibilité d'ajouter les caractères de formatage, hormis les accolades. Ces caractères sont spécifiques à chaque type d'objet. Le tableau suivant nous donne les caractères utilisées pour les chaînes des caractères, les entiers et les réels :

TYPE D'OBJET	CARACTÈRE
Entier (`int`)	`d`
Réel (`float`)	`f`
Chaîne de caractères (`str`)	`s`

Table 1.2 – Caractères de formatage en Python

Exemple 1.6.5.

```
In [1]: x = 2850
In [2]: y = 44389
In [3]: z = (x - 300)/y
In [4]: print("Ce nombre est: ", z)
Ce nombre est: 0.057446664714231
```

On remarque qu'il y a plusieurs décimaux après la virgule, pour le réduire à 2 ou 4, on utilise l'instruction `.format(: .xf)`) avec x un entier positif représentant le nombre de chiffre après la virgule.

Exemple 1.6.6.

```
In [5]: print("Ce nombre est {:.2f}".format(z))
Ce nombre est 0.06
In [6]: print("Ce nombre est {:.4f}".format(z))
Ce nombre est 0.0574
```

On peut aussi préciser combien de caractères vous voulez qu'un résultat soit écrit et comment se fait l'alignement (à gauche, à droite ou centré). Nous pouvons l'expérimenter avec les ligne de code de l'exemple suivant :

Exemple 1.6.7.

```
In [1]: print(40); print(40000)
40
40000
```

```
In [2]: print("{:>6d}".format(40)); print("{:>6d}".format(40000))
40
40000

In [3]: print("{:<6d}".format(40)); print("{:<6d}".format(40000))
40
40000

In [4]: print("{:^6d}".format(40)); print("{:^6d}".format(40000))
40
40000

In [5]: print("{:*^6d}".format(40)); print("{:*^6d}".format(40000))
**40**
40000*

In [6]: print("{:0>6d}".format(40)); print("{:0>6d}".format(40000))
000040
040000
```

Un programme écrit avec des commentaires permet la clarté dans l'écriture d'un programme. Cela permet aussi à un autre programmeur de continuer l'écriture du programme sans beaucoup de complications. En Python, le commentaire se met toujours après le symbole #.

En effet,tout ce qui est compris entre le symbole # et ce saut de ligne est ignoré. Un commentaire peut donc occuper la totalité d'une ligne (on place le # en début de ligne) ou une partie seulement, après une instruction (on place le # après la ligne de code pour la commenter plus spécifiquement).

Exemple 1.6.8.

```
In [1]: #Ceci est un commentaire
```

1.6.3 Lecture à partir du clavier

Jusqu'ici, nous avons seulement afficher les éléments à sans donner la possibilité à l'utilisateur d'entrer ses propres valeurs à partir du clavier. Python nous donne aussi la possibilité de demander à l'utilisateur d'entrer ses propres valeurs à partir du clavier. Pour ce faire, la syntaxe utilisée est est la suivante :

`nomVariable = input(["msg"])`

où `"msg"` est le message d'attente optionnel qui indique à l'utilisateur ce qu'il doit entrer.

Exemple 1.6.9.

```
In  [1]: nom = input("Entrer votre nom: ")

Entrer votre nom: Kamingu

In  [2]: sexe = input("Entrer votre sexe: ")

Entrer votre sexe: Masculin

In  [3]: print("{0} est de sexe {1}".format(nom, sexe))
    ...:
Kamingu est de sexe Masculin
```

Nota 1.6.1. *Il sied de signaler que la fonction* ***input()*** *renvoie toujours une chaîne de caractères. Si on souhaite que l'utilisateur entre une valeur numérique, on doit donc faire le transtypage (qui sera donc de toute façon de type string) permettant de convertir la chaîne de caractères en une valeur numérique du type qui convient.*

Exemple 1.6.10.

```
In [1]: age = input("Entrer votre age S.V.P. ")

Entrer votre age S.V.P. 25

In [2]: type(age)
Out[2]: str

In [3]: ageN = int(age)

In [4]: type(ageN)
Out[4]: int
```

Exercices

Exercice 1.1.

On a besoin de calculer le produit intérieur brut (PIB) par la formule (l'approche dépenses) suivante :

$$PIB = C + I + G + (X–M), \tag{1.1}$$

avec C la consommation, I les investissements, G les dépenses publiques, X l'exportation et M l'importation. Écrire un programme Python qui demande à l'utilisateur de fournir les informations relatives au calcul du PIB en utilisation l'approche 1.1.

Exercice 1.2.

Le calcul de la valeur future V_n au bout du temps n connaissant le capital (ou valeur initiale disponible aujourd'hui) C peut se calculer de deux manières. On peut utiliser la formule 1.2 ou la formule 1.3.

$$V_n = C(1 + n\rho), \tag{1.2}$$

appelée valeur future à intérêt simple où ρ est le taux d'intérêt et $(1 + n\rho)$ est appelé facteur d'accumulation.

$$V_n = C(1 + \rho)^n, \tag{1.3}$$

appelée valeur future à intérêt composé où ρ est son taux d'intérêt et $(1 + \rho)^n$ son facteur d'accumulation.
Écrire un programme Python qui demande à l'utilisateur les différents paramètres (C, n et ρ) dans différents cas et calcule le facteur d'accumulation ainsi que les valeurs futures.

Exercice 1.3.

Dans un fichier nommé taux.py, calculez un un taux donnée par : `taux` = `(925.256 - 45)/325`. Ensuite, affichez le contenu de la variable `taux` à l'écran avec 0, 1, 2, 3, 4 puis 5 décimales sous forme arrondie en utilisant `format()`. On souhaite que le programme affiche la sortie suivante :

```
Ce taux a pour valeur 3 %

Ce taux a pour valeur 2.7 %

Ce taux a pour valeur 2.71 %

Ce taux a pour valeur 2.708 %

Ce taux a pour valeur 2.7085 %

Ce taux a pour valeur 2.70848 %
```

Exercice 1.4.

La puissance électrique que l'on note souvent par P et qui a pour unité le watt (symbolisé par W) est définie comme le produit de la tension électrique (que l'on note par U) aux bornes (en volts, dont le symbole est V) et de l'intensité du courant électrique (que l'on note par I) qui le traverse (en ampères, dont le symbole est A). Écrire un programme Python qui demande à l'utilisateur d'entrer les valeurs de U et I et calcule ensuite la valeur de puissance électrique.

Chapitre 2

STRUCTURES DE CONTRÔLE

2.1 Structures conditionnelles

On appelle ***structure conditionnelle*** *une structure de contrôle dont on a l'alternative entre une ou plusieurs instructions. Ces structures conditions nous permettent d'exécuter une ou plusieurs instructions dans un cas, et d'autres instructions dans un autre cas.*

Pour ce faire, Python dispose d'instructions capables de tester une certaine condition et de modifier le comportement du programme en conséquence.

2.1.1 Instruction `if : ... else : ...`

Exemple 2.1.1.

```
In[1]: if nombre > 0: #C'est-a-dire nombre positif
  ...:   print("C'est un nombre positif.")
  ...: else: #C'est-a-dire nombre non-positif
  ...:   print("C'est un nombre negatif.")
  ...:
C'est un nombre positif.
```

L'expression placée entre parenthèses après `if` est ce que nous appellerons une *condition*. L'instruction `if` permet de tester la validité de cette condition. Si la condition est vraie, alors l'instruction que nous avons indentée après le `:` est exécutée, tandis que l'instruction `else` (≪sinon≫, en anglais) permet de programmer une exécution alternative.

Opérateurs de comparaison

Généralement, les conditions sont formées de deux termes (`terme1` et `terme2`) séparés par un opérateur de comparaison (`opComp`) ; d'où le format (`terme1 opComp terme2`). Les opérateurs de comparaison utilisés pour les numériques sont repris dans le tableau suivant :

OPÉRATEUR	SIGNIFICATION LITTÉRALE
<	Strictement inférieur à
>	Strictement supérieur (e) à
<=	Inférieur (e) ou égal à
>=	Supérieur (e) ou égal à
==	Égal (e) à
! =	Différent (e) de

Table 2.1 – Opérateurs de comparaison

2.1.2 Instruction `if : ... elif : ... [elif : ...] else : ...`

Le présent schéma des structures conditionnelles est une généralisation du précédent, car il ne prévoit pas qu'un seul ou des actions à effectuer, il y a plusieurs actions.

Exemple 2.1.2. *Soit à écrire un programme qui détermine si un nombre entier est positif, négatif ou nul, on aura le code suivant :*

```
In [1]: nombre = input("Entrer le nombre de votre choix: ")

Entrer un nombre au choix: -245

In [2]: nombreConv = int(nombre) #Pour convertir le string nombre en integer nombreConv

In [3]: if nombreConv > 0:
   ...:     print("Le nombre est positif")
        elif nombreConv == 0:
   ...:     print("Le nombre est nul")
        else: #Le cas restant, donc nombre<0
   ...:     print("Le nombre est negatif")
   ...:
Le nombre est negatif
```

Opérateurs logiques

Il existe aussi des conditions composées. Ces conditions se forment au moyen des opérateurs de comparaisons et des opérateurs logiques (ou booléens).

Ci-après les opérateurs logiques que dispose Python :

Exemple 2.1.3. *Soit à écrire un programme Python qui permette la saisie des scores de quatre candidats au premier tour des élections législatives nationales, dans le District de Mont-Amba. Ce programme traitera ensuite le candidat numéro 1 (et uniquement lui) : il dira s'il est élu, battu, s'il se trouve en ballottage favorable (il participe au second tour en étant arrivé en tête à l'issue du premier tour) ou défavorable (il participe au second tour sans avoir été en tête au premier tour). Voici la règle qu'obéissent ces élections :*

MOTS-CLÉS UTILISÉS	TYPE	SIGNIFICATION
and	n-aire	ET logique
or	n-aire	OU logique
not	1-aire	NON logique

Table 2.2 – Opérateurs logiques

— *lorsque l'un des candidats obtient plus de 50% des suffrages, il est élu dès le premier tour ;*
— *en cas de deuxième tour, peuvent participer uniquement les candidats ayant obtenu au moins 12,5% des voix au premier tour.*

On a le code suivant :

```
In [1]: scoreC1 = input("Entrer le score du premier candidat: ")

Entrer le score du premier candidat: 41

In [2]: scoreC2 = input("Entrer le score du deuxieme candidat: ")

Entrer le score du deuxieme candidat: 12

In [3]: scoreC3 = input("Entrer le score du troisieme candidat: ")

Entrer le score du troisieme candidat: 20

In [4]: scoreC4 = input("Entrer le score du quatrieme candidat: ")

Entrer le score du quatrieme candidat: 27

In [5]: sc1 = float(scoreC1)

In [6]: sc2 = float(scoreC2)

In [7]: sc3 = float(scoreC3)

In [8]: sc4 = float(scoreC4)

In [9]: if sc1 > 50:
   ...:     print("Elu au premier tour")
        elif sc2 > 50 or sc3 > 50 or sc4 > 50 or not(sc1 >= 12,5):
   ...:     print("Battu, sorti de la course!!!")
        elif sc1 >= sc2 and sc1 >= sc3 and sc1 >= sc4:
   ...:     print("Ballotage favorable")
        else:
   ...:     print("Ballotage defavorable")

Ballotage favorable
```

2.2 Structures itératives ou boucles

*On appelle **boucle**, appelée aussi **structure répétitive** ou **structure itérative** ou tout simplement **itérations** est une structure de contrôle qui permet de répéter l'exécution d'instruction ou d'un bloc d'instructions.*

2.2.1 Boucle `while`

La boucle `while` permet de répéter un bloc d'instructions tant qu'une condition est vraie (`while` signifie «tant que» en anglais). Cette structure de contrôle est importante lorsque le nombre de répétitions n'est pas prédéterminé.

La syntaxe de cette structure est la suivante :

```
while condition:
       bloc dInstructions
```

Exemple 2.2.1.

```
In [1]: compteur = 1

In [2]: while compteur < 10:
   ...:      print("On a la valeur", compteur)
   ...:    compteur = compteur*2
   ...:    print("Fin")
   ...:
On a la valeur 1
On a la valeur 2
On a la valeur 4
On a la valeur 8
Fin
```

2.2.2 Boucle `for`

La boucle `for` travaille sur des séquences. Elle est en fait spécialisée dans le parcours d'une séquence de plusieurs données. Cette structure de contrôle est importante lorsque le nombre de répétitions est prédéterminé.

```
for element in sequence:
     bloc dInstructions
```

Exemple 2.2.2.

```
In [1]: for i in range(8):
   ...:     print("On a la valeur", i)
   ...:
On a la valeur 0
On a la valeur 1
On a la valeur 2
On a la valeur 3
On a la valeur 4
On a la valeur 5
On a la valeur 6
On a la valeur 7
```

2.2.3 Transformation d'une boucle `for` en une boucle `while`

Comme nous l'avions explicité dans les lignes ci-haut, la boucle `for` est utilisée lorsque le nombre de répétitions est prédéterminée (i.e. connue en avance), tandis que la boucle `while` est utilisée lorsque le nombre de répétitions n'est pas connu en avance. Néanmoins, nous pouvons transformer d'une boucle `for` en boucle `while`.

Exemple 2.2.3.

```
In [1]: compteur = 0

In [2]: while compteur < 5:
   ...:     print("On a la valeur", compteur)
   ...:     compteur = compteur + 1
   ...:
On a la valeur 1
On a la valeur 2
On a la valeur 3
On a la valeur 4
```

Exemple 2.2.4.

```
In [1]: for compteur in range(5)
   ...:     print("On a la valeur", compteur)
   ...:
On a la valeur 1
On a la valeur 2
On a la valeur 3
On a la valeur 4
```

2.3 Instructions `break` et `continue`

Ces deux instructions permettent de modifier le comportement d'une boucle (`for` ou `while`) avec un test. Les instructions `break` et `continue` permettent respectivement de stopper la boucle et de sauter à l'itération suivante.

Exemple 2.3.1.

```
In [1]: for i in range(50):
   ...:     if i == 20:
   ...:         break
   ...:     print(i)
   ...:
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

Nous remarquons que si le programme atteint l'index 20 (sachant qu'une liste commence toujours par 0), la boucle stoppe l'exécution de l'instruction `print(i)`.

Exemple 2.3.2.

```
In [2]: for i in range(12):
   ...:     if i == 8:
   ...:         continue
   ...:     print(i)
   ...:
0
1
2
3
4
5
6
7
9
10
11
```

Nous remarquons que si le programme atteint l'index égale à 8, l'instruction `print(i)` (c'est-à-

dire l'affichage de 8) est sautée.

2.4 La clause `else`

Lorsque nous incluront une boucle dans un programme Python, il peut arriver que la boucle se termine par épuisement d'une séquence (cas de la boucle `for`) ou que la condition de boucle devienne fausse (cas de la boucle `while`), mais pas interrompue par une instruction `break`. Dans les deux cas précités, l'utilisation de la clause `else` est recommandée.

Le mot-clé `else` existe aussi pour les boucles et s'utilise en association avec le mot-clé `break`.

Exercices

Exercice 2.1.

Écrire un programme Python qui demande à l'utilisateur son nom et son sexe (M ou F). En fonction de ces données, afficher ≪Cher Monsieur≫ ou ≪Chère Mademoiselle≫ suivi du nom de la personne.

Exercice 2.2.

Écrire un programme Python qui demande deux nombres à l'utilisateur et l'informe ensuite si produit est négatif ou positif (on laisse de côté le cas où le produit est nul) sans calculer le produit des deux nombres.

Exercice 2.3.

Écrire un programme Python qui permet de résoudre l'équation du second degré de la forme $ax^2 + bx + c = 0$ dans $\mathbb{R}$.

Exercice 2.4.

Écrire un programme Python qui demander à l'utilisateur d'entrer trois longueurs a, b, c. À l'aide de ces trois longueurs, déterminer s'il est possible de construire un triangle. Le programme doit être capable de déterminer ensuite si ce triangle est rectangle, isocèle, équilatéral ou quelconque. Attention : un triangle rectangle peut être isocèle.

Exercice 2.5.

Écrire un algorithme Python qui demande le pourcentage d'un étudiant à l'utilisateur. Ensuite, il l'informe de sa mention :
— ≪Plus grande distinction≫, de 90 à 100 ;
— ≪Grande distinction≫, de 80 à 89 ;
— ≪Distinction≫, de 70 à 79 ;
— ≪Satisfaction≫, de 50 à 69 ;
— ≪Ajourné≫, de 40 à 49 ;
— ≪Refusé≫ moins de 40.

Exercice 2.6.

Ecrire un programme Python de facturation qui accomplit les actions suivantes :

— Le programme lit les libellés, les quantités et les prix unitaires de trois produits quelconque ;

— Le programme calcule le total partiel de chaque produit, le total général et le net à payer sachant qu'une remise de 5% est accordée sur le montant total si le total est au moins à 4500 unités monétaires.

Exercice 2.7.

Ecrire un programme Python de facturation qui lit les libellés, les quantités, les prix unitaires de trois produits quelconque et le taux TVA. Puis le programme calcule le total partiel de chaque produit, le total général et le prix total taxe comprise.

Exercice 2.8.

Écrire un programme qui demande à l'utilisateur une annee et ensuite déterminer si cette année est bissextile ou non. Pour rappel, une année est bissextile si cette année est divisible par 4. Elle ne l'est cependant pas si cette année est un multiple de 100, à moins que cette annee ne soit multiple de 400.

Exercice 2.9.

Écrire un programme Python qui demande à l'utilisateur d'entrer un nombre entre 1 et 3. Si l'utilisateur choisit 1, le programme affichera "Vous avez choisi un : le premier, l'unique, l'unité ..." ; si il choisit 2, le programme affichera "Vous préférez le deux : la paire, le couple, le duo ..." et si il choisit 3,le programme affichera "Vous optez pour le plus grand des trois : "le trio, la trinité, le triplet ...", dans tous les autres cas, le programme affichera "Un nombre entre UN et TROIS, s.v.p.".

Exercice 2.10.

Écrire un programme Python qui affiche la table de multiplication d'un nombre n par m.

Exercice 2.11.

Écrire un programme Python qui demande un nombre compris entre 40 et 100, jusqu'à ce que la réponse `rep` convienne. En cas de réponse `rep` qui convienne, on apparaître un message : `"Le nombre rep convient"` . En cas de réponse `rep` est supérieure à 100, on fera apparaître un message : `"Le nombre rep est trop grand!"`, et inversement, `"Le nombre rep est trop petit!"` si le nombre est inférieur à 40.

Exercice 2.12.

Un nombre premier est un entier naturel (1 étant exclus) qui admet exactement deux diviseurs distincts entiers et positifs qui sont alors 1 et lui-même. Écrire un Programme Python qui teste si un nombre quelconque n entré au clavier est premier ou non

Chapitre 3

CHAÎNES DE CARACTÈRES, LISTES, TUPLES ET DICTIONNAIRES

Jusqu'ici, nous avons traité les types simples (entier, réel, etc.), mais dans ce chapitre, nous allons traiter les types composites ou les conteneurs. Ces conteneurs sont des types d'objets qui contiennent les autres types d'objets. En Python, les conteneurs les plus utilisés sont (*i*) les chaînes de caractères, (*ii*) les listes (*iii*) les tuples et (*iv*) les dictionnaires.

3.1 Chaînes de caractères

3.1.1 Déclaration d'une chaîne de caractères

Dans un programme Python, on peut déclarer une chaîne de caractères, soit par des apostrophes (simple quotes), soit par des guillemets (double quotes). Les chaînes de caractères sont des objets de type `string`.

Exemple 3.1.1.

```
In [1]: ch1 = "Papa"

In [2]: ch2 = 'Maman'
```

3.1.2 Opérations sur les chaînes de caractères

A. Indiçage

De même pour les listes, nous pouvons accéder à un caractère d'une chaîne à partir de son *indice* (ou *index*) de la liste.

Comme pour les listes, les éléments d'une séquence sont toujours indicés (ou numérotés) de la même manière, c'est-à-dire à partir de 0.

Exemple 3.1.2.

```
In [1]: ch1 = "Tombola Muke"

In [2]: ch1[0]
Out[2]: 'T'

In [3]: ch1[5]
Out[3]: 'l'

In [4]: ch1[7]
Out[4]: ' '
```

B. Concaténation et duplication

Comme nous l'avions vu plutôt, on utilise l'opérateur + pour la concaténation, ainsi que l'opérateur ∗ pour la duplication.

Exemple 3.1.3.

```
In [1]: prenom = "Roger"

In [2]: nom = "Emone"

In [3]: prenomNom = prenom + " " + nom

In [4]: prenomNom
Out[4]: 'Roger Emone'

In [4]: nom*2
Out[4]: 'EmoneEmone'
```

C. Fonction `len()`

L'instruction `len()` permet de connaître la longueur ou le nombre de caractères d'une chaine.

Exemple 3.1.4.

```
In [1]: len("")
Out[1]: 0

In [2]: phrase = "Je suis mathematicien"

In [3]: len(phrase)
Out[3]: 21
```

D. Extraction de fragments de chaînes

Il peut arrive qu'on extraire une petite chaîne d'une chaîne plus longue. Python propose pour cela une technique simple que l'on appelle ***slicing*** (« découpage en tranches ») [Swi12]. Cette technique consiste à indiquer entre crochets les indices correspondant au début et à la fin de la « tranche » que l'on souhaite extraire.

Exemple 3.1.5.

```
In [1]: nom = "Gloria Batoka"

In [2]: morceau1 = print(nom[0:4])
Glor

In [2]: morceau1 = print(nom[2:7])
oria
```

Nota 3.1.1. *Les indices de découpage ont des valeurs par défaut : un premier indice non défini est considéré comme* 0*, tandis que le second indice omis prend par défaut la taille de la chaîne complète. [Swi12]*

Exemple 3.1.6.

```
In [1]: nom = "Gloria Batoka"

In [2]: print(nom[:4]) #Affichage de 4 premiers caracteres
Glor

In [3]: print(nom[4:]) #Affichage tout ce qui suit les 4 premiers caracteres
ia Batoka
```

3.2 Listes

3.2.1 Définition (Liste)

*On appelle **liste** une structure de données qui permet de regrouper de manière structurée une série de valeurs.*

Le grand avantage avec Python est qu'une liste peut contenir des valeurs de type différent (par exemple entier et chaîne de caractères), ce qui leur confère une grande flexibilité.

3.2.2 Déclaration d'une liste

La déclaration d'une liste se fait par une série de valeurs, séparées par des virgules, et l'ensemble étant enfermé dans des crochets. Une liste vide se note donc [].

Sa syntaxe est :

```
nomListe = [item1, item2, ... , itemn]
```

Exemple 3.2.1.

```
In [1]: animaux = ["Lion", "Abeille", "Tigre", "Fourmi"]

In [2]: joursDates = ['lundi', 'mardi', 'mercredi', 2018, 'jeudi', 'vendredi', 1990]

In [3]: nombres = [2.0, 51, 4j, complex(2.0, 85.2)]

In [4]: print(animaux)
['Lion', 'Abeille', 'Tigre', 'Fourmi']

In [5]: print(joursDates)
['lundi', 'mardi', 'mercredi', 2018, 'jeudi', 'vendredi', 1990]

In [6]: print(nombres)
[2.0, 51, 4j, (2+85.2j)]
```

Nota 3.2.1. *On peut aussi declarer une liste en utilisant la fonction* `list(chaineDeCaracteres)`, *pour obtenir la liste* `['c', 'h', 'a', 'i', 'n', 'e', 'D', 'e', 'C', 'a', 'r', 'a', 'c', 't', 'e', 'r', 'e', 's']`

Exemple 3.2.2.

```
In [1]: maListe = list("Herve Kamango")

In [2]: taListe = list("123456")

In [3]: maListe
Out[3]: ['H', 'e', 'r', 'v', 'e', ' ', 'K', 'a', 'm', 'a', 'n', 'g', 'o']

In [4]: taListe
Out[4]: ['1', '2', '3', '4', '5', '6']
```

3.2.3 Opérations sur les listes

A. Indiçage

On peut appeler les éléments d'une liste par leur position. Ce numéro est appelé *indice* (ou *index*) de la liste.

Il sied de noter que les indices d'une liste de n éléments commence à 0 et se termine à $n-1$.

Exemple 3.2.3.

```
In [67]: cetteListe = ["Poisson", 2.05, "Papa", 5j, 53, "Herman", "Armel"]

In [68]: cetteListe[0]
Out[68]: 'Poisson'

In [69]: cetteListe[5]
Out[69]: 'Herman'
```

B. Concaténation et duplication

En Python, on peut utiliser l'opérateur + pour la concaténation, ainsi que l'opérateur ∗ pour la duplication.

Exemple 3.2.4.

```
In [10]: jours1 = ["Dimanche", "Lundi", "Mardi", "Mercredi"]

In [11]: jours2 = ["Jeudi", "Vendredi", "Samedi"]

In [12]: semaine = jours1 + jours2

In [13]: print(semaine)
['Dimanche', 'Lundi', 'Mardi', 'Mercredi', 'Jeudi', 'Vendredi', 'Samedi']

print(jours2*2)
['Jeudi', 'Vendredi', 'Samedi', 'Jeudi', 'Vendredi', 'Samedi']
```

C. Fonction `len()`

L'instruction `len()` permet de connaître la longueur ou le nombre d'éléments.

Exemple 3.2.5.

```
In [8]: len(animaux)
Out [8]: 4
```

D. Fonction `append()`

L'instruction `.append(element)` est une fonction qui permet d'ajouter un élément `element` à la fin de la liste.

Exemple 3.2.6.

```
In [15]: etudiants = ["Acacia", "Madiata", "Daniel", "Prince"]

In [16]: etudiants.append("Percide")

In [17]: print(etudiants)
['Acacia', 'Madiata', 'Daniel', 'Prince', 'Percide']
```

E. Fonction `del()`

L'instruction `del(nomList[index])` est une fonction qui permet de supprimer l'élément `nomList[index]`.

Exemple 3.2.7.

```
In [8]: noms = ["Cedrick", "Tsasa", "Lokota", "Emone", "Kiala", "Yannick"]
In [9]: del(noms[3])
In [10]: print(noms)
['Cedrick', 'Tsasa', 'Lokota', 'Kiala', 'Yannick']
```

F. Fonctions `list()` and `range()`

L'instruction `range()` est une fonction qui permet de de générer des nombres entiers compris dans un intervalle lorsqu'elle est utilisée en combinaison avec la fonction `list()`.

Pour l'utiliser, il existe trois schémas.

F.1. Schéma `list(range(n))`

Dans ce schéma, le programme va générer une liste contenant tous les nombres entiers de 0 inclus à n exclus.

Exemple 3.2.8.

```
In [1]: list(range(14))
Out [1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

F.2. Schéma `list(range(d, f))`

Dans ce schéma, le programme va générer une liste contenant tous les nombres entiers entre le début d inclus et la fin f exclus.

Exemple 3.2.9.

```
In [51]: list(range(14,25))
Out[51]: [14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
```

F.3. Schéma `list(range(d, f, p))`

Dans ce schéma, le programme va générer une liste contenant tous les nombres entiers entre le début d inclus et la fin f exclus, avec le pas d'incrémentation ou de décrémentation p.

Exemple 3.2.10.

```
In [54]: list(range(230,1870, 200))
Out[54]: [230, 430, 630, 830, 1030, 1230, 1430, 1630, 1830]
```

3.3 Tuples

3.3.1 Notion

Un **tuple** est un conteneur se définissant comme une liste, sauf qu'on utilise comme délimiteur des parenthèses au lieu des crochets. À la différence des listes, les tuples, une fois créés, ne peuvent être modifiés : on ne peut plus y ajouter d'objet ou en retirer.

3.3.2 Déclaration d'un tuple

Pour créer un tuple, on enferme ses éléments (séparées par des virgules) dans une paire de parenthèses ( ). La syntaxe reste valide même si les parenthèses sont omises. Nous remarquons qu'un tuple se définit comme une liste, sauf qu'on utilise comme délimiteur des parenthèses (ou rien) au lieu des crochets.

Sa syntaxe est :

```
nomListe = (item1, item2, ... , itemn)
```

Exemple 3.3.1.

```
In [1]: tup1 = 25, "Papa", 2.08, "Je suis"

In [2]: tup2 = (24, "Maman", 3.08, "Tu es")

In [3]: tup1
Out[3]: (25, 'Papa', 2.08, 'Je suis')

In [4]: tup1 = 25, "Papa", 2.08, "Je suis"
Out[4]: (24, 'Maman', 3.08, 'Tu es')
```

3.3.3 Opérations sur les tuples

A. Fonction `len()`

L'instruction `len()` permet de connaître la longueur ou le nombre d'éléments.

Exemple 3.3.2.

```
In  [7]: monTuple = ("Mangue", 20.2, "Tomate", 25, "Choux",52.31, "Piment", 12)

In  [8]: len(monTuple)
Out [8]: 8
```

B. Fonctions `tuple()` and `range()`

Comme pour les listes, l'instruction `range()` est une fonction qui permet de de générer des nombres entiers compris dans un intervalle lorsqu'elle est utilisée en combinaison avec la fonction `tuple()`. Donc, la fonction `tuple()` fonctionne exactement comme la fonction `list()`.

Exemple 3.3.3.

```
In [1]: tuple(range(14))
Out[1]: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)

In [2]:tuple(range(5, 23))
Out[2]: (5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22)

In [3]:tuple(range(24, 52, 8))
Out[3]: (24, 32, 40, 48)

In [4]: tuple("ABCDEFGHIJKL")
Out[4]: ('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L')
```

3.4 Dictionnaires

3.4.1 Notion

Un des types des données utilisés par Python est le *dictionnaire. Le* ***dictionnaire*** *est une collection non ordonnée d'objets, c'est à dire qu'il n'y a pas de notion d'ordre (ou encore pas d'indice).*

Ce type d'objets est utilisé par Python pour représenter diverses fonctionnalités : on peut par exemple retrouver les attributs d'un objet grâce à un dictionnaire particulier. L'accès aux **valeurs** d'un dictionnaire par des **clés**. L'exemple suivant va illustrer la notion de dictionnaire, de clé et de valeur.

La différence majeure qui existe entre les listes et les dictionnaires est dans l'ordre des éléments. Comme dans une liste, les éléments mémorisés dans un dictionnaire peuvent être de n'importe quel type, c'est-à-dire, ils peuvent être des valeurs numériques, des chaînes, des listes, des dictionnaires, et même aussi des fonctions, des classes ou des instances.

3.4.2 Déclaration d'un dictionnaire

Il existe deux manières de créer un dictionnaire. Nous pouvons commencer par créer un dictionnaire vide, puis le remplir petit à petit. Du point de vue de la syntaxe, on reconnaît un dictionnaire au fait que ses éléments sont enfermés dans une paire d'accolades. Un dictionnaire vide sera donc noté { }.

Exemple 3.4.1.

```
In [70]: dico = {}

In [71]: dico["Pen"] = "Stylo"

In [72]: dico["Book"] = "Livre"

In [73]: dico["Pencil"] = "Crayon"

In [74]: dico["Board"] = "Tableau"

In [75]: print(dico)
{'Pen': 'Stylo', 'Pencil': 'Crayon', 'Book': 'Livre', 'Board': 'Tableau'}
```

La deuxième manière consiste à mettre les éléments la forme d'une série d'éléments séparés par des virgules, le tout étant enfermé entre deux accolades.

Exemple 3.4.2.

```
In [80]: dico2 = {"nom": "Mbikayi", "poids": 82, "taille": 1.80, "age":29, "profession
    ": "etudiant-chercheur"}

In [81]: print(dico2)
{'poids': 82, 'nom': 'Mbikayi', 'profession': 'etudiant-chercheur', 'age': 29, 'taille
    ': 1.8}
```

3.4.3 Opérations sur les dictionnaires

A. Récupération de la valeur d'une clé donnée

Pour récupérer la valeur d'une clé donnée, il suffit d'utiliser une syntaxe `dictionnaire["cle"]`.

Exemple 3.4.3.

```
In [89]: dico["Pen"]
Out[89]: 'Stylo'
```

B. Affichage des clés et des valeurs d'un dictionnaire

Pour afficher les clés et les valeurs d'un dictionnaire (sous forme de liste), on utilise les méthodes `keys()` et `values()`. On peut aussi afficher les couples clés – valeurs en utilisant la méthode `items`.

Exemple 3.4.4.

```
In [80]: dico.keys()
Out[80]: dict_keys(['Pen', 'Pencil', 'Book', 'Board'])

In [81]: dico.values()
Out[81]: dict_values(['Stylo', 'Crayon', 'Livre', 'Tableau'])

In [82]: dico.items()
Out[82]: dict_items([('Pen', 'Stylo'), ('Pencil', 'Crayon'), ('Book', 'Livre'), ('
    Board', 'Tableau')])
```

C. Test d'existence d'une clé

Pour tester l'existence d'une clé, on utilise le mot-clé `in`.

Exemple 3.4.5.

```
In [80]: if "age" in dico2:
    ...: print("Voila ce que je veux!")
    ...:
Voila ce que je veux!
```

D. Conversion des dictionnaires en listes ou en tuples

Python nous donne la possibilité de convertir un dictionnaire en liste ou en tuple respectivement par les fonctions `list()` et `tuple()`.

Exemple 3.4.6.

```
In [87]: list(dico)
Out[87]: ['Pen', 'Pencil', 'Book', 'Board']

In [88]: tuple(dico)
Out[88]: ('Pen', 'Pencil', 'Book', 'Board')
```

Exercices

Exercice 3.1.

Écrire un programme Python qui demande à l'utilisateur une lettre (on exclut les lettre accentuées) et qui dit si la lettre est une voyelle ou non.

Exercice 3.2.

Écrire un programme Python qui demande à une chaîne de caractères et l'affiche dans l'ordre inverse.

Exercice 3.3.

Écrire un programme Python qui demande à une chaîne de caractères et le découpe en fragments de 5 caractères chacun.

Exercice 3.4.

Écrire un programme Python qui demande à l'utilisateur une chaîne de caractères et ensuite convertit tous les caractères minuscules en majuscules, et vice-versa (dans une phrase fournie en argument).

Exercice 3.5 (Énigme du père Fouras)**.**

A Fort Boyard, le père Fouras nous pose l'énigme suivante :
« Pour ouvrir le coffre où se trouve la clé, trouve la combinaison à trois chiffres sachant que :
— Le nombre est inférieur à 200.
— Deux de ses chiffres sont identiques.
— La somme de ses chiffres est égale à 5.
— C'est un nombre pair. »
On se propose d'utiliser une méthode dite brute force, c'est à dire d'utiliser une boucle et à chaque itération on teste les 4 conditions. Écrire un programme Python qui résout cet énigme.

Exercice 3.6.

Créer une liste A contenant quelques éléments. Effectuer une vraie copie de cette liste dans une nouvelle variable B. Suggestion : créez d'abord une liste B de même taille que A mais ne contenant que des zéros. Remplacer ensuite tous ces zéros par les éléments tirés de A.

Exercice 3.7.

Écrire un programme Python qui permet d'afficher les 10 premiers nombres paires en utilisant les fonctions `list()` et `range()`.

Exercice 3.8.

Écrire un programme Python qui permet d'afficher les 10 premiers nombres impaires en utilisant les fonctions `list()` et `range()`.

Exercice 3.9.

Écrire un programme Python qui recherche le plus petit et le plus grand éléments présents dans une liste donnée.

Exercice 3.10.

Écrire un programme Python qui analyse un par un tous les éléments d'une liste de nombres (par exemple celle de l'exercice précédent) pour générer deux nouvelles listes. L'une contiendra seulement les nombres pairs de la liste initiale, et l'autre les nombres impairs.

Exercice 3.11.

Soient les listes suivantes :

```
liste1 = [31,28,31,30,31,30,31,31,30,31,30,31]
liste2 = ["Janvier", "Février", "Mars", "Avril", "Mai", "Juin", "Juillet", "Août",
"Septembre", "Octobre", "Novembre", "Décembre"]
```

Écrire un programme Programme qui insère dans la seconde liste tous les éléments de la première, de telle sorte que chaque nom de mois soit suivi du nombre de jours correspondant : `['Janvier',31, 'Février',28, 'Mars',31, etc.]` .

Exercice 3.12.

Un nombre premier est un nombre qui n'est divisible que par un et par lui-même. Écrire un programme qui établit la liste de tous les nombres premiers compris entre 1 et 1000, en utilisant la méthode du crible d'Eratosthène :

- Créer une liste de 1000 éléments, chacun initialisé à la valeur 1.
- Parcourir cette liste à partir de l'élément d'indice 2 : si l'élément analysé possède la valeur 1, mettre à zéro tous les autres éléments de la liste, dont les indices sont des multiples entiers de l'indice auquel vous êtes arrivé.
  Lorsque on aurait parcouru ainsi toute la liste, les indices des éléments qui seront restés à 1 seront les nombres premiers recherchés.
  En effet : A partir de l'indice 2, on annule tous les éléments d'indices pairs : 4, 6, 8, 10, etc.
  Avec l'indice 3, on annule les éléments d'indices 6, 9, 12, 15, etc., et ainsi de suite. Seuls resteront à 1 les éléments dont les indices sont effectivement des nombres premiers.

Exercice 3.13.

Écrire un programme Python qui défini un dictionnaire dont les clés sont des numéros (de 0 à 10) en chiffres et les valeurs sont des numéros en lettres. Ensuite le programme doit échanger les clés et les valeurs d'un dictionnaire.

Exercice 3.14.

Écrire un programme Python qui définit deux dictionnaires Français – Anglais : l'un pour les jours de la semaine et l'autre pour les mois de l'année. Le programme doit afficher les differentes cles et les differentes valeurs de ces deux dictionnaires respectifs, sous forme

(i) des listes ;

(ii) des tuples.

Chapitre 4

BIBLIOTHÈQUES SCIENTIFIQUES

Les modules sont des programmes Python qui contiennent des fonctions que l'on est amené à réutiliser souvent (on les appelle aussi *bibliothèques* ou *libraries*, en anglais). L'un des avantages de Python est son panoplie des bibliothèques qu'il incorpore.

Dans ce chapitre, nous allons voir quelques bibliothèques scientifiques. Ces bibliothèques contiennent des fonctions que les scientifiques (économistes, statisticiens, data scientists, mathématiciens, ingénieurs) utilisent le plus souvent dans le cadre de leurs applications (traitement et analyse des données, calculs trigonométriques, représentations graphiques des fonctions, etc.).

4.1 Généralités sur les bibliothèques

4.1.1 Importation de bibliothèques

Python offre de très nombreuses bibliothèques de fonctions prédéfinies pour réaliser des tâches scientifiques courantes. Ainsi, Python nous offre deux possibilités d'importer une bibliothèques prédéfinies.

Possibilité 1

La première possibilité consiste à importer la bibliothèque et ensuite accéder de manière indirecte à une fonction de ladite bibliothèque.

Sa syntaxe est la suivante :

```
import nomDeLaBibiotheque

nomDeLaBibiotheque.nomDeLaFonction([parametre(s)])
```

Exemple 4.1.1.

Supposons que nous voulions utiliser la fonction **sin** de la bibliothèque **math** [1]

```
In [1]: import math

In [2]: math.sin(80)
Out[2]: -0.9938886539233752
```

Remarquons que la première ligne du code, c'est-à-dire **import math** importe le module **math**,

1. Nous le verrons en détail plus tard.

mais les fonctions du module ne sont pas accessibles directement. Il faut alors utiliser le préfixe `math.` pour désigner la fonction. Cette méthode a pour avantage l'utilisation des modules différents qui contiennent chacun une fonction du même nom.

Possibilité 2

Cette possibilité consiste à importer une fonction en utilisant le mot-clé **`from`**. Dans cette méthode, on n'utilise pas le préfixe comme on l'a fait dans la méthode précédente.

Sa syntaxe est la suivante :

```
from nomDeLaBibiotheque
import nomDeLaFonction
```

Puis, on peut utiliser la fonction directement.

Exemple 4.1.2.

```
In [1]: from math import sin

In [2]: sin(80)
Out[2]: -0.9938886539233752
```

On peut aussi utiliser toutes les fonctions de la bibliothèque en utilisant la syntaxe suivante :

```
from nomDeLaBibiotheque import*
```

Exemple 4.1.3.

```
In [1]: from math import *

In [2]: sin(80)
Out[2]: -0.9938886539233752

In [3]: cos(80)
Out[3]: -0.11038724383904756
```

4.1.2 Obtention d'aide sur les bibliothèques importées

Pour obtenir de l'aide sur un module rien de plus simple, il suffit d'invoquer la commande `help()`. En effet, La commande `help()` est une commande plus permettant d'avoir de l'aide sur n'importe quel objet chargé en mémoire. La syntaxe pour l'obtention d'aide est la suivante :

```
help(nomObjetEnMemoire)
```

Nota 4.1.1. *Pour demander de l'aide dans le cas d'une bibliothèque, il faut d'abord l'importer pour éviter le message d'erreur.*

4.2 Bibliothèque `math`

Le module ***math*** est une bibliothèque scientifique de Python permettant d'effectuer opérations mathématiques usuelles (fonctions logarithmiques, fonctions exponentielles, fonctions racines d'indice n, fonctions trigonométriques inverses, etc.).

Voici quelques fonctions de la bibliothèque `math` de Python :

FONCTIONS	EXPLICATIONS
`math.floor(x)`	Renvoie la partie entière de x.
`math.ceil(x)`	Renvoie un entier immédiatement supérieur à x.
`math.log(x)`	Renvoie le logarithme de x en base naturelle.
`math.log10(x)`	Renvoie le logarithme de x en base 10.
`math.log(x, y)`	Renvoie le logarithme de x en base y.
`math.exp(x)`	Renvoie l'exponentielle de x.
`math.sqrt(x)`	Renvoie le racine carrée de x.
`math.pow(x, y)`	Renvoie la x puissance y.
`math.factorial(x)`	Renvoie la factorielle x.
`math.gamma(x)`	Renvoie la fonction gamma au point x.
`math.sin(x)`	Renvoie le sinus de l'angle x.
`math.cos(x)`	Renvoie le cosinus de l'angle x.
`math.tan(x)`	Renvoie le tangente de x.
`math.`π	Renvoie la constante π.
`math.`e	Renvoie l'exponentielle e.

Table 4.1 – Quelques fonctions de la bibliothèque `math`

Exemple 4.2.1.

```
In [1]: from math import*

In [2]: factorial(25)
Out[2]: 15511210043330985984000000

In [3]: sin(60)
Out[3]: -0.3048106211022167

In [4]: pow(5,3)
Out[4]: 125.0

In [5]: pi
Out[5]: 3.141592653589793

In [6]: e
Out[6]: 2.718281828459045
```

4.3 Bibliothèque `fractions`

Le module ***fractions*** est une bibliothèque scientifique permettant d'effectuer des calculs sur des nombres rationnels (nombres pouvant être écrits sous forme de fraction).

Cependant, il existe cinq façons de construire une fonction en utilisant la bibliothèque `fractions`, on a :

A. fractions.Fraction(numerateur, denominateur)

Dans la première façon, nous introduisons le numérateur et le dénominateur de notre fraction, sachant que le dénominateur est toujours différent de 0, sinon on a le message `ZeroDivisionError`.

Exemple 4.3.1.

```
In [1]: from fractions import Fraction

In [2]: Fraction(35, 78)
Out[2]: Fraction(35, 78)

In [3]: Fraction(52, 24)
Out[3]: Fraction(13, 6)

In [4]: Fraction(0, 5)
Out[4]: Fraction(0, 1)

In [5]: Fraction(32, 0)
Traceback (most recent call last):

File "<ipython-input-5-4e087caf85c8>", line 1, in <module>
  Fraction(32, 0)

File "/usr/lib/python3.5/fractions.py", line 186, in __new__
  raise ZeroDivisionError('Fraction(%s, 0)' % numerator)

ZeroDivisionError: Fraction(32, 0)
```

B. fractions.Fraction(uneAutreFraction)

Dans la deuxième façon, l'argument est une fraction. La valeur renvoyée est la même que celle de `uneAutreFraction`.

Exemple 4.3.2.

```
In [1]: from fractions import Fraction

In [2]: quotien = Fraction(21, 53)

In [3]: quotien
Out[3]: Fraction(21, 53)

In [4]: Q = Fraction(quotien)

In [5]: Q
Out[5]: Fraction(21, 53)
```

C. fractions.Fraction(unFloat)

Dans cette version, il suffit d'écrire le nombre en décimal. Mais il sied de signaler qu'étant donnée les problèmes de virgules flottantes binaires, le fraction ne renvoie pas généralement ce qu'on s'attend (voir exemple 4.3.3), d'où l'intérêt de spécifier la limite du dénominateur avec `textbflimit_denominator(nombreMax)`, avec `nombreMax` la limite maximale.

Exemple 4.3.3.

```
In [1]: from fractions import*

In [2]: from math import*

In [3]: Fraction(2.25)
Out[3]: Fraction(9, 4)

In [4]: Fraction(0.125)
Out[4]: Fraction(1, 8)

In [5]: Fraction(1.1)
Out[5]: Fraction(2476979795053773, 2251799813685248)

In [6]: Fraction('3.1415926535').limit_denominator(100000)
Out[6]: Fraction(208341, 66317)

In [7]: Fraction(cos(pi/3))
Out[7]: Fraction(4503599627370497, 9007199254740992)

In [8]: Fraction(cos(pi/3)).limit_denominator(100)
Out[8]: Fraction(1, 2)
```

D. fractions.Fraction(unDecimal)

Cette version requiert qu'on ait un décimal. Sur ce, on utilise la bibliothèque `decimal`.

Exemple 4.3.4.

```
In [1]: from fractions import*

In [2]: from decimal import Decimal

In [3]: from math import*

In [4]: Fraction(Decimal(1.50))
Out[4]: Fraction(3, 2)
```

E. fractions.Fraction(unString)

La dernière version du constructeur de fraction requiert un string ou une instance Unicode.

Exemple 4.3.5.

```
In [1]:from fractions import Fraction

In [2]:Fraction("14/32")
Out[2]:Fraction(7, 16)

In [3]:Fraction("32.560")
Out[3]:Fraction(814, 25)

In [4]:Fraction('-.225')
Out[4]:Fraction(-9, 40)
```

Nota 4.3.1. *La bibliothèque* `fractions` *possède une fonction* `fractions.gcd(x, y)` *permettant de renvoyer le plus grand commun diviseur (pgcd) de deux nombres entiers x et y.*

Exemple 4.3.6.

```
In [1]: from fractions import*

In [2]: gcd(5,35)
Out[2]: 5

In [3]: gcd(int(2.05), ceil(4.2))
Out[3]: 1
```

4.4 Bibliothèque `random`

Le module ***random*** est une bibliothèque scientifique de Python permettant de générer des nombres pseudo-aléatoires, en utilisant des distributions des probabilités (loi binomiale, loi normale, etc.).

Les fonctions les plus utilisées de cette bibliothèque sont reprises dans le tableau suivant :

FONCTIONS	EXPLICATIONS
`random.random()`	Renvoie un nombre réel dans l'intervalle 0.0 et 1.0 exclus.
`random.randint(x, y)`	Choix aléatoire d'un nombre entier compris entre x et y.
`random.choice(Seq)`	Choix aléatoire d'un élément se trouvant dans la séquence `Seq`.
`random.shuffle(Seq [, random])`	Mélange des éléments d'une séquence `Seq`.
`random.sample(pop, k)`	Renvoie une liste d'éléments uniques de longueur k choisi dans la séquence ou l'ensemble de la population `pop`. Il est utilisé pour échantillonnage aléatoire sans remise.
`random.uniform(x, y)`	Renvoie un nombre réel compris entre x et y pour $a \leq b$.
`random.gauss(mu, sigma)`	Loi normale de paramètres `mu` (la moyenne) et `sigma` (l'écart-type).
`random.expovariate(lambd)`	Loi exponentielle négative de paramètres `lambd` (la moyenne).

Table 4.2 – Quelques fonctions de la bibliothèque `random`

Exemple 4.4.1.

```
In [1]: from random import*

In [2]: random()
Out[2]: 0.40636555987715084

In [3]: uniform(1, 20)
Out[3]: 17.014790108160774

In [4]: choice('Yves Mbolo')
Out[4]: 'v'

In [5]:sample([1, 2, 3, 4, 5, 6, 7, 8], 4)
Out[5]: [5, 2, 3, 8]
```

4.5 Bibliothèque `statistics`

Le module ***statistics*** est une bibliothèque scientifique de Python permettant d'effectuer des calculs des statistiques des données numériques.

Les fonctions les plus utilisées de cette bibliothèque sont reprises dans le tableau suivant :

FONCTIONS	EXPLICATIONS
`statistics.means(Seq)`	Moyenne arithmétique des données de la séquence `Seq`.
`statistics.harmonic_mean(Seq)`	Moyenne harmonique des données de la séquence `Seq`.
`statistics.median(Seq)`	Médiane des données de la séquence `Seq`.
`statistics.mode(Seq)`	Mode (ou valeur dominante) des données de la séquence `Seq`.
`statistics.stdev(Seq)`	Écart-type des données de la séquence `Seq`.
`statistics.variance(Seq)`	Variance des données de la séquence `Seq`.

Table 4.3 – Quelques fonctions de la bibliothèque `statistics`

Exemple 4.5.1.

```
In [16]: from statistics import *

In [17]: mean([25, 25, 35, 45])
Out[17]: 32.5

In [18]: from fractions import Fraction as F

In [19]: variance([F(4,5), F(3,8), F(1,6)])
Out[19]: Fraction(4501, 43200)
```

4.6 Bibliothèque `numpy`

Le module ***numpy*** (de l'anglais *Numerical Python* (en français Python numérique)) est une bibliothèque numérique de Python permettant d'effectuer plusieurs opérations sur les tableaux multidimensionnels, et des opérations mathématiques de haut niveau (fonctions spéciales, algèbre linéaire, statistiques, etc.) opérant sur ces tableaux.

4.6.1 Importation de la bibliothèque

Comme pour toutes les autres bibliothèques, nous importons la bibliothèque `numpy` de la manière suivante :

```
In [1]: import numpy as np
```

4.6.2 Construction d'un(e) tableau (ou matrice)

Contrairement à ce que nous avons vu sur les listes, les tableaux `numpy` ne peuvent contenir des membres que d'un seul type. Ce type est automatiquement déduit au moment de la création du tableau, et a un impact sur les opérations qui y seront appliquées. On peut aussi spécifier le type manuellement.

A. Construction d'un(e) tableau (matrice) avec la fonction `array`

Pour créer un tableau avec `numpy`, on peut utiliser la fonction `array`.

Exemple 4.6.1.

```
In [1]: import numpy as np

In [2]: x = np.array([1, 2, 3, 4])

In [3]: x
Out[3]: array([1, 2, 3, 4])

In [4]: y = np.array([1, 2, 3, 4], dtype='int')

In [5]: y
Out[5]: array([1, 2, 3, 4])

In [6]: z = np.array([[1,2,-2],[3,0,4],[-2,5,6]])

In [7]: z
Out[7]:
array([[ 1, 2, -2],
       [ 3, 0, 4],
       [-2, 5, 6]])

In [8]: z = w = np.array([[[1,2],[2,52]],[[4,44],[45,8]]])

In [9]: w
Out[9]:
array([[[ 1, 2],
        [ 2, 52]],

       [[ 4, 44],
        [45, 8]]])
```

B. Construction automatique d'un(e) tableau (matrice)

La bibliothèque `numpy` contient plusieurs fonctions donnant la possibilité, surtout pour les tableaux larges, de les créer directement.

Les fonctions les plus utilisés de cette bibliothèque sont reprises dans le tableau suivant :

FONCTIONS	EXPLICATIONS
`arange(d, f [, p])`	Permet de créer une matrice à partir des entiers de d à f exclus, avec un pas p. Lorsque p n'est pas défini, alors la valeur par défaut est 1.
`linspace(d, f, n)`	Permet de créer un tableau de n valeurs, espacées uniformément entre d et f.
`zeros(T, type)`	Permet de créer une matrice nulle de taille T et de type `type`.
`ones(T, type)`	Permet de créer une matrice unité de taille T et de type `type`.
`full(T, nbre, type)`	Permet de créer une matrice nulle de taille T et de type `type`.
`eye(n)`	Permet de créer une matrice identité d'ordre n.

Table 4.4 – Quelques fonctions de creation d'un tableau `numpy`

Exemple 4.6.2.

```
In [1]: import numpy as np

In [2]: x = np.arange(10)

In [3]: x
Out[3]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]) # Un tableau commencant a 0 et qui se
    termine a 9, avec un pas de 1

In [4]: y = np.arange(20, 50, 3)

In [5]: y
Out[5]: array([20, 23, 26, 29, 32, 35, 38, 41, 44, 47]) # Un tableau des elements compris
     entre 20 et 50 exclus de pas 3

In [6]: z = np.linspace(2, 4, 4)

In [7]: z
Out[5]: array([ 2. , 2.5, 3. , 3.5, 4. ]) # Un tableau de 5 valeurs, espacees uniformement
    entre 2 et 4
```

Exemple 4.6.3.

```
In [1]: import numpy as np

In [2]: np.zeros(5, int) # Un tableau de longueur 5, rempli d'entiers qui valent 0
Out[2]: array([0, 0, 0, 0, 0])

In [3]: np.zeros((4, 4), float) # Un tableau de taille 4x4 rempli de nombres a virgule
    flottante de valeur 0
Out[3]:
array([[ 0., 0., 0., 0.],
       [ 0., 0., 0., 0.],
       [ 0., 0., 0., 0.],
       [ 0., 0., 0., 0.]])

In [4]: np.ones(6, int) # Un tableau de longueur 6, rempli d'entiers qui valent 0
Out[4]: array([1, 1, 1, 1, 1, 1])

In [5]: np.ones((4, 5), float) # Un tableau de taille 4x5 rempli de nombres a virgule flottante
    de valeur 1
Out[5]:
array([[ 1., 1., 1., 1., 1.],
       [ 1., 1., 1., 1., 1.],
       [ 1., 1., 1., 1., 1.],
       [ 1., 1., 1., 1., 1.]])

In [6]: np.full((4, 5), 10.14) # Un tableau 3x5 rempli de 10.14
Out[6]:
array([[ 10.14, 10.14, 10.14, 10.14, 10.14],
       [ 10.14, 10.14, 10.14, 10.14, 10.14],
       [ 10.14, 10.14, 10.14, 10.14, 10.14],
       [ 10.14, 10.14, 10.14, 10.14, 10.14]])

In [5]: np.eye(4) # La matrice identite de taille 4x4
Out[5]:
array([[ 1., 0., 0., 0.],
       [ 0., 1., 0., 0.],
       [ 0., 0., 1., 0.],
       [ 0., 0., 0., 1.]])
```

4.6.3 Opérations sur les tableaux avec `numpy`

Exercices

Exercice 4.1.

Ecrire un programme Python qui demande à l'utilisateur la base a et un nombre x et ensuite calcule le logarithme du nombre x en base a, c'est-à-dire $\log_a x$.

Exercice 4.2.

Soient deux champs électriques (notés E_1 et E_2 et exprimés en newtons par coulomb (N/C)) dont les directions sont en sens opposé. La résultante de ces champs seront calculer par

$$E = \sqrt{E_1^2 + E_2^2 + 2E_1E_2\cos\theta}.$$

Écrire un programme Python qui demande à l'utilisateur les modules de ces deux champs électriques et l'angle formé par ces champs électrique et ensuite calcule la résultante de ces champs.

Exercice 4.3.

A l'aide d'un programme Python, générer une séquence aléatoire de 6 chiffres, ceux-ci étant des entiers tirés entre 1 et 4. Utiliser la bibliothèque `random` et la fonction `randint()`.

Exercice 4.4.

Écrire un programme Python qui permet de calculer le sinus, le cosinus, la tangente et la cotangente de $\frac{\pi}{2}$ en utilisant la bibliothèque `math` et la constante `pi`.

Exercice 4.5.

Écrire un programme Python qui demande à l'utilisateur un nombre angle en dégrée et ensuite calcule le sinus, le cosinus, la tangente et la cotangente de l'angle.

Exercice 4.6.

Soit un cercle de rayon 1 (en rouge) inscrit dans un carré de coté 2 (en bleu). Avec $R = 1$, l'aire du carré vaut $(2R)^2$ soit 4 et l'aire du cercle vaut πR^2 soit π.

En choisissant N points aléatoires (à l'aide d'une distribution uniforme) à l'intérieur du carré, la probabilité que ces points se trouvent aussi dans le cercle est

$$p = \frac{\text{aire du cercle}}{\text{aire du carré}} = \frac{\pi}{4}. \tag{4.1}$$

Soit n, le nombre points effectivement dans le cercle, il vient alors

$$p = \frac{n}{N} = \frac{\pi}{4}, \tag{4.2}$$

Fig XXXXXXXXXXXXXXXXXXXX Cercle de rayon 1 inscrit dans un carré de côté 2

d'où

$$4 \times \frac{n}{N}. \tag{4.3}$$

Déterminer une approximation de π par cette méthode. Pour cela, pour N itérations :

— Choisir aléatoirement les coordonnées x et y d'un point entre -1 et 1. Utiliser la fonction `uniform()` de la bibliothèque `random`.

— Calculer la distance entre le centre du cercle et ce point. Pour rappel, si $A(x_1, y_1)$ et $B(x_2, y_2)$ sont deux points, la distance d entre A et B est calculée de la manière suivante :

$$d(A, B) = \sqrt{(x_2 - x_1)^2 - (y_2 - y_1)^2}. \tag{4.4}$$

— Déterminer si cette distance est inférieure au rayon du cercle, c'est-à-dire si le point est dans le cercle ou pas.

— Si le point est effectivement dans le cercle, incrémenter le compteur n.

Finalement calculer le rapport entre n et N et proposer une estimation de π. Quelle valeur de π obtient-on pour 100 itérations ? 1000 itérations ? 10000 itérations ? Comparer les valeurs obtenues à la valeur de π fournie par la bibliothèque `math`.

Chapitre 5

PROCÉDURES ET FONCTIONS

L'un des concepts les plus importants en programmation est celui de **sous-programme**. D'une part, les sous-programmes permettent en effet de décomposer un programme complexe en une série de sous-programmes plus simples, lesquels peuvent à leur tour être décomposés en fragments plus petits, et ainsi de suite.

D'autre part, les sous-programmes épargnent le programmeur à réaliser plusieurs fois la même opération au sein d'un programme. De plus, elles sont réutilisables, c'est-à-dire on peut l'utiliser dans les programmes sans pouvoir le réécrire.

Nous avons déjà utilisé les sous-programmes dans les chapitres précédents : ce sont des fonctions prédéfinies. Mais dans le cadre de ce chapitre, nous allons traiter les fonctions personnalisées, c'est-à-dire des fonctions définies par le programmeur.

5.1 Généralités sur les sous-programmes

5.1.1 Définition (Sous-programme)

Un sous-programme, en programmation, est un programme qui est associe à un autre appelée ***programme principal****.*

5.1.2 Types des sous-programmes

Il existe en effet deux types de sous-programme, en occurrence la **procédure** et la **fonction**. La différence entre une procédure et une fonction est qu'une fonction doit en effet renvoyer une valeur lorsqu'elle se termine.

Dans le cadre de ce cours, on parlera simplement de *fonction* pour faire allusion à soit la procédure, soit la fonction, car contrairement à certains langages de programmation qui utilisent des instructions différentes pour définir les fonctions et les procédures, Python utilise la même instruction pour définir les unes et les autres.

5.1.3 Déclaration d'une fonction

Pour définir une fonction, on utilise le mot-clé `def` suivi du nom de la fonction ; ensuite suivent des parenthèses qui contiennent les éventuels paramètres de la fonction puis le caractère : qui délimite le début d'une séquence de code. On a la syntaxe suivant :

```
def nomDeLaFonction(listeDeParametres):
    ...
    bloc d'instructions
    ...
```

Où `listeDeParametres` spécifie quelles informations il faudra fournir en guise d'arguments lorsque l'on voudra utiliser cette fonction (les parenthèses peuvent parfaitement rester vides si

la fonction ne nécessite pas d'arguments).

Pour nommer une fonction, on utilise n'importe quel nom, à l'exception des mots réservés du langage (voir Chapitre 1, et à la condition de n'utiliser aucun caractère spécial ou accentué (le caractère souligné « _ » est permis). Comme c'est le cas pour les noms de variables, il vous est conseillé d'utiliser surtout des lettres minuscules, notamment au début du nom (les noms commençant par une majuscule seront réservés aux classes que nous étudierons au Chapitre 6).

Nota 5.1.1. *Dans un programme Python, la définition des fonctions doit précéder leur utilisation.*

Exemple 5.1.1. *Supposons qu'on veuille afficher les* 10 *premiers multiples d'un nombre. Nous pouvons utiliser les lignes de code suivant :*

```
In [1]: #Definition des fonctions

In [2]: def multiple(nombre):
    ...: n = 1
    ...:  while n <11 :
    ...:       print(n*nombre, end =' ')
    ...:       n = n + 1
    ...:

In [3]: #Programme principal

In [4]: multiple(14)
14 28 42 56 70 84 98 112 126 140
```

Dans l'exemple ci-dessus, la valeur que nous indiquons entre parenthèses lors de l'appel de la fonction (et qui est donc un argument) est automatiquement affectée au paramètre `nombre`. Dans le corps de la fonction, `nombre` joue le même rôle que n'importe quelle autre variable. Lorsque nous entrons la commande `multiple(14)` , nous signifions à la machine que nous voulons exécuter la fonction `multiple()` en affectant la valeur 14 à la variable `nombre`.

Nota 5.1.2. *Il peut arriver que la liste des paramètres soient vide. Et cela n'empêche pas que la fonction ne fonctionne.*

Exemple 5.1.2.

```
In [1]: #Definition des fonctions

In [2]: def fin():
   ...:    print("Il est temps, que chacun rentre chez soi!")

In [3]: #Programme principal

In [4]: fin()
Il est temps, que chacun rentre chez soi!
```

5.2 Définition de la valeur renvoyée par la fonction

Pour définir ce que doit être la valeur renvoyée par la fonction (au sens strict), on utilise l'instruction `return`.

Cette instruction permet de *renvoyer une valeur* lorsqu'elle se termine. Une fonction (au sens strict) peut donc s'utiliser à la droite du signe d'affectation.

Exemple 5.2.1. *Dans cet exemple, nous définissons la fonction* `somme()` *permettant de renvoyer la somme de deux nombre.*

```
In [1]: #Definition des fonctions

In [2]: def somme(n1, n2):
...: return n1+n2
...:

In [3]: #Programme principal

In [4]: nombre1 = 3521

In [5]: nombre2 = 253.254

In [6]: S = somme(nombre1, nombre2)

In [7]: print("La somme de ces nombres est: ", S)
Out[7]: La somme de ces nombres est: 3774.254
```

5.3 Portée des objets dans un programme modulaire

Lorsqu'on définit les variables dans un programme Python, elles sont placées par l'interpréteur dans un des deux dictionnaires représentant le contexte d'exécution [Zia09] :

(i) Le premier dictionnaire contient l'ensemble des variables **globales** ou **publiques** et est accessible par le biais de la primitive `globals()`. Dans ce cas, il s'agit des variables qui sont définies à l'extérieur du corps d'une fonction, ces variables sont accessibles et modifiables depuis n'importe quelle ligne de code de ce programme, sauf dans la fonction ;

(ii) Le second, contient l'ensemble des variables **locales** ou **privées** et est accessible par le biais de la primitive `locals()`. Dans ce cas, il s'agit des variables qui sont définies à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même.

Exemple 5.3.1. *Reprenons l'exemple où l'on suppose vouloir afficher les* 10 *premiers multiples d'un nombre. Nous pouvons utiliser les lignes de code suivant :*

```
In [1]: #Definition des fonctions

In [2]: def multiple(nombre):
    ...: n = 1 #Variable locale, car declaree dans la fonction
    ...: while n <11 :
    ...:       print(n*nombre, end =' ')
    ...:       n = n + 1
    ...:

In [3]: #Programme principal

In [4]: multiple(3)
3 6 9 12 15 18 21 24 27 30
```

Dans l'exemple ci-haut, il y aura message d'erreur si nous exécutons l'instruction suivante :

```
 In [5]: y = 4 + n

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-df6ca2354cee> in <module>()
----> 1 y = 4 + n

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

Ce message est affiché pour le simple fait que la variable n est locale, et par conséquent ne peut pas utilisée dans n'importe quelle partie du programme.

Exemple 5.3.2. *Modifions 5.3.1, en changeant la position de la déclaration de la variable n, on a :*

```
In [1]: n = 1 #Variable globale, car declaree dans la fonction

In [2]: #Definition des fonctions

In [3]: def multiple(nombre):
    ...:  while n <11 :
    ...:        print(n*nombre, end =' ')
    ...:        n = n + 1
    ...:

In [4]: #Programme principal

In [5]: multiple(3)

---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-22-7fa38b39f662> in <module>()
1 #Programme principal
----> 2 multiple(3)

<ipython-input-21-09db88c698ba> in multiple(nombre)
2 def multiple(nombre):
3
----> 4    while n < 11:
5         print(n*nombre, end = ' ')
6         n = n + 1

UnboundLocalError: local variable 'n' referenced before assignment
```

Ce message est affiché pour le simple fait que la variable n est globale, et par conséquent ne peut pas être modifiée dans la fonction. Il faudrait donc créer une variable locale qui prendra la valeur de n.

```
In [1]: n = 1 #Variable globale, car declaree dans la fonction

In [2]: #Definition des fonctions

In [3]: def multiple(nombre):
    ...:  m = n
    ...:  while m <11 :
    ...:        print(m*nombre, end =' ')
    ...:        m = m + 1
    ...:

In [4]: #Programme principal

In [5]: multiple(3)
```

5.4 Fonctions récursives

5.4.1 Définition (Fonction récursive)

*Une fonction est dite **récursive** lorsqu'elle est définie en fonction d'elle-même, i.e. lorsqu'elle s'appelle elle-même.*

Exemples 5.4.1. *(a) La fonction factoriel définie de la manière suivante :*

$$n! = \begin{cases} 1 & si\ n <= 1, \\ n * (n-1)! & sinon. \end{cases}$$

(b) La fonction d'Ackermann définie de la manière suivante :

$$A(m,n) = \begin{cases} n+1 & si\ m = 0, \\ A(m-1,1) & si\ m > 0\ et\ n = 0, \\ A(m-1, A(m, n-1)) & sinon. \end{cases}$$

5.4.2 Principe et intérêt d'une fonction récursive

Dans la definition d'une fonction récursive, on doit :

(1) décomposer un problème en un ou plusieurs sous-problèmes du même type. On résout les sous-problèmes par des appels récursifs ;

(2) la décomposition doit en fin de compte conduire à un cas élémentaire qui lui n'est pas décomposé en sous-problème. Ce cas élémentaire est appelé **condition d'arrêt**.

Exemple 5.4.1.

(1) Problème : Calculer $n!$.
Sous-problème : Calculer $(n-1)!$. On multipliera le résultat par n. Le calcul de $(n-1)!$ passe par le calcul de $(n-2)!$. Le calcul de $(n-2)!$ passe par le calcul de $(n-3)!$, et ainsi de suite.

(2) À la fin, le problème devient le calcul de $0!$ qui est le critère d'arrêt (cas élémentaire), qui a pour résultat 1.

Ainsi, la récursivité permet d'écrire des programmes concis et élégants.

Le code Python suivant permet de définir la fonction factoriel de manière récursive :

```
In [1]: def factoriel(n):
  ...: if n<1: #Cas elementaire ou critere d'arret
  ...:       return 1
  ...: else:
  ...:     return n*factoriel(n - 1) #Appel recursive
  ...:

In [2]: #Programme principal

In [3]: factoriel(0)
Out[3]: 1

In [4]: factoriel(3)
Out[4]: 6

In [5]: factoriel(6)
Out[5]: 720
```

Exercices

Exercice 5.1.

Écrire et tester une fonction (en Python) permettant de calculer le périmètre d'un rectangle en connaissant les mesures de ces côtés.

Exercice 5.2.

Écrire et tester une fonction (en Python) permettant de calculer la surface d'un rectangle en connaissant les mesures de ces côtés.

Exercice 5.3.

La puissance électrique (mesurée en watts (W)) est donnée par la formule suivante :

$$P = U \cdot I. \tag{5.1}$$

Avec U la tension électrique mesurée en volts (V) et I l'intensité du courant électrique mesurée en ampères (A).

Écrire et tester une fonction (en Python) permettant de calculer la puissance électrique d'un générateur.

Exercice 5.4.

Écrire et tester une fonction (en Python) permettant de déterminer si un nombre est positif, négatif ou nul.

Exercice 5.5.

Écrire et tester une fonction (en Python) permettant de savoir si un nombre nommé *nbre* est divisible par un autre nommé *nom*.

Exercice 5.6.

Écrire et tester une fonction récursive (en Python) pour calculer la somme :

$$u_n = 1 + 2^4 + 3^4 + 4^4 + \cdots + n^4.$$

Exercice 5.7 (Puissance n d'un nombre x)**.**

La puissance n d'un nombre x peut se definir recursivement de la manière suivante :

$$x^n = \begin{cases} 1 & \text{si } n = 0, \\ x * x^{n-1} & \text{si } n > 0. \end{cases}$$

Écrire et tester une fonction récursive (en Python) qui demande à l'utilisateur un nombre entier x et sa puissance n et ensuite calcule la valeur de x^n

Exercice 5.8 (Suite de Fibonacci)**.**

La suite de Fibonacci est une suite créee par Leonardo Fibonacci pour étudier la reproduction des lapins. Cette suite est definie de la manière suivante : les deux premiers termes sont des 1, et tous les autres termes sont trouvées en calculant la somme de deux termes qui les précédent. Mathématiquement :

$$F(n) = \begin{cases} 1 & \text{si } n = 1 \text{ et } n = 2, \\ F(n-1) + F(n-2) & \text{si } n > 2. \end{cases}$$

Écrire et tester une fonction récursive (en Python) qui demande à l'utilisateur un nombre entier n et calcule les n premiers termes de la suite de Fibonacci.

Exercice 5.9 (Algorithme d'Euclide)**.**

Écrire et tester une fonction récursive (en Python) qui demande à l'utilisateur deux nombres entiers positifs et calcule son plus grand commun diviseur (pgcd) en utilisant l'Algorithme d'Euclide. Pour rappel, l'algorithme d'Euclide permet de calculer le pgcd en utilisant la fonction récursive suivante :

$$pgcd(a, b) = \begin{cases} a & \text{si } b = 0, \\ pgcd(b, a \mod b) & \text{sinon.} \end{cases}$$

Exercice 5.10.

Un nombre de Fermat est un nombre de la forme $2^{2^n} + 1$. Ces nombres obéissent à la relation de récurrence suivante :

$$F(n) = \begin{cases} 3 & \text{si } n = 0, \\ (F(n-1) - 1)^2 & \text{sinon.} \end{cases}$$

Écrire et tester une fonction récursive (en Python) qui demande à l'utilisateur un nombre entier n et ensuite calcule la valeur de $F(n)$.

Exercice 5.11 (Fonction d'Ackermann)**.**

La fonction d'Ackermann est définie de la manière suivante :

$$A(m, n) = \begin{cases} n + 1 & \text{si } m = 0, \\ A(m-1, 1) & \text{si } m > 0 \text{ et } n = 0, \\ A(m-1, A(m, n-1)) & \text{sinon.} \end{cases}$$

Écrire et tester une fonction récursive (en Python) qui demande à l'utilisateur deux nombres entiers m et n et ensuite calcule la valeur de $A(m, n)$.

Exercice 5.12 (Combinaisons)**.**

La combinaison de n éléments pris p à p peut se calculer en utilisant la relation de Pascal définie de la manière suivante

$$C_n^p = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n, \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{sinon.} \end{cases}$$

Écrire et tester une fonction récursive (en Python) qui demande à l'utilisateur la valeur de n et p, avec $n \leq p$et ensuite calcule la valeur de C_n^p.

Chapitre 6

PROGRAMMATION ORIENTÉE-OBJETS AVEC PYTHON

Les premiers langages de programmation n'incluaient pas l'orienté objet. A titre d'illustration, le langage C n'utilise pas ce concept et il aura fallu attendre le C++ pour utiliser la puissance de l'orienté objet dans une syntaxe proche de celle du C. Le langage BASIC n'utilise pas ce concept et il aura fallu attendre le Visual BASIC pour utiliser la puissance de l'orienté objet dans une syntaxe proche de celle du BASIC. De même pour le langage Pascal, il aura fallu attendre le Delphi pour utiliser la puissance de l'orienté objet dans une syntaxe proche de celle du Pascal.

6.1 Généralités sur la programmation orientée-objet

6.1.1 Objet

Un objet est une collection des données et de comportements associes.

Un objet n'est pas seulement quelque chose de tangible, que nous pouvons toucher, palper, sentir, mais c'est un modèle de quelque chose qui peut faire certaine chose et dont certaines choses peuvent se faire sur lui.

Exemple 6.1.1. ***L'etudiant Cyrille****, **l'assistant Yves*** *et* ***le stylo Bravo*** *sont des objets.*

6.1.2 Classe

Une classe est une description abstraite (condensée) d'un ensemble d'objets du domaine de l'application : elle définit leur structure, leur comportement et leurs relations.

De ce qui précède, on remarque qu'un objet est **issu** d'une classe ou tout simplement c'est une **instance** d'une classe.

Exemple 6.1.2. ***Etudiant****, **Assistant*** *et* ***Stylo*** *sont des classes.*

6.1.3 Attribut

Une classe correspond à un concept global d'information et se compose d'un ensemble d'informations élémentaires, appelées attributs de classe. Cet attribut représente la modélisation d'une information élémentaire, c'est-á-dire d'une caractéristique représentée par son nom et son format.

Exemple 6.1.3. *Dans la classe Etudiant, les attributs peuvent être :* ***matricule, nom, post-nom, date de naissance, sexe****,* ***promotion****, qui ont comme format respectivement : chaîne de caractères, chaîne de caractères, chaîne de caractères, date et caractère.*

Un attribut particulier, qui permet de repérer de manière unique chaque objet, instance de la classe est appelé **identifiant**.

Exemple 6.1.4. *Dans la classe Etudiant, l'attribut* ***matricule*** *est un identifiant, car il permet de repérer de manière unique les objets de la classe Etudiant. En effet, chaque étudiant possède un numéro matricule propre á lui.*

6.1.4 Méthode

La méthode ou opération représente un élément de comportement des objets, défini de manière globale dans la classe.
Une méthode est une fonctionnalité assurée par une classe. La description des opérations peut préciser les paramètres d'entrée et de sortie ainsi que les actions élémentaires à exécuter.

Exemple 6.1.5. *Dans la classe Etudiant, les actions suivantes peuvent etre considérées comme des méthodes :* ***Apprendre, Reviser, Lire***

6.1.5 Héritage

L'héritage est un principe selon lequel on peut créer une classe á partir d'une classe existante appelée ***super-classe*** *ou* ***classe mère*** *ou encore* ***classe parente****.*

6.1.6 Programmation orientée-objet

6.2 Paradigme orienté-objet avec Python

6.2.1 Classes en Python

Et effet, le langage Python est totalement orienté objet : en Python, tout est objet. En réalité, une variable, une liste, un tuple, un module, une fonction, ..., sont des objets, qui sont, en principe des instances des classes.

Exercices

Chapitre 7

INTERFACES GRAPHIQUES AVEC PYTHON

7.1 Introduction sur les interfaces graphiques

7.1.1 Définition (Interface graphique)

*Une **interface graphique** (ou **GUI**, Graphical User Interface) est un dispositif, qui présente des informations de façon visuelle et reçoit des messages via une souris (ou autre pointeur), destiné à permettre les interactions entre programme et utilisateur.*

Pour faire des applications avec des interfaces graphiques, Python dispose de plusieurs bibliothèques. Les plus utilisées sont les bibliothèques suivantes : Tkinter, wxPython, pyQT, pyGTK, etc. Hormis ces bibliothèques, il existe également des possibilités d'utiliser les bibliothèques de widgets Java et les MFC de Windows.

Dans le cadre de ce cours, nous nous appesantirons cependant sur **Tkinter**, dont il existe fort heureusement des versions similaires (et gratuites) pour les systèmes d'exploitation tels que Linux, Windows et Mac OS.

En effet, Tkinter est une bibliothèque libre, gratuit, portable dans différentes systèmes d'exploitation (Linux, Windows et Mac OS), orientée objets et facile à utiliser.

7.1.2 Importance d'une interface graphique

Plusieurs raisons sont à épingler pour signaler l'importance de l'utilisation d'une interface graphique, à savoir :

— l'utilisation utilisation agréable du programme ;
— les résultats du programme sont plus immédiatement parlants ;
— la mise en évidence de l'effet d'un paramètre, ...
— les interfaces graphiques sont accessibles aux personnes peu familières avec les ordinateurs.

7.2 Présentation de Tkinter

Tkinter est une abréviation de **Tool kit Interface**. C'est une bibliothèque intégrée Python. Tkinter offre un moyen de créer des interfaces graphiques via Python. L'avantage de Tkinter est qu'il est disponible par défaut, sans nécessiter une installation supplémentaire (surtout pour les distributeurs Windows). Un autre avantage est que les interfaces créées sur Tkinter dans un système peut être portable dans un autre système.

7.3 Premier exemple

Dans ce premier exemple, nous allons créer une petite interface avec Tkinter. Nous allons utiliser le procédé suivant :

(1) On commence par importer Tkinter. Cette importation permet donc àl'utiliser.

(2) On crée ensuite un objet de la classe `Tk`. La plupart du temps, cet objet sera la fenêtre principale de notre interface ;

(3) On crée un `Label`, c'est-à-dire un objet graphique affichant du texte ;

(4) On appelle la méthode `pack` de notre `Label`. Cette méthode permet de positionner l'objet dans notre fenêtre (et, par conséquent, de l'afficher).

(5) Enfin, on appelle la méthode `mainloop` de notre fenêtre racine. Cette méthode ne retourne que lorsqu'on ferme la fenêtre.

Exemple 7.3.1.

```
In [1]: from tkinter import *

In [2]: fenetre = Tk()

In [3]: label = Label(fenetre, text="Voici mon premier interface graphique!")

In [4]: label.pack()

In [5]: fenetre.mainloop()
```

Après exécution de ce code, le fenêtre suivante apparaît :

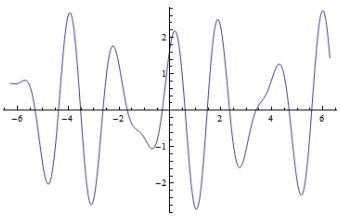

Figure 7.1 – Premier Exemple de l'interface Tkinter.

7.4 Widgets Tkinter

Pour pouvoir créer des logiciels avec interface graphiques, nous utilisons des éléments graphiques telles que `Label` et autres (boutons, champs de texte, cases à cocher, barres de progression...). Ces éléments graphiques sont appelés des **widgets**.À savoir que, pour qu'un widget apparaisse, il faut :

(i) qu'il prenne, en premier paramètre du constructeur, la fenêtre principale ;

(ii) qu'il fasse appel à la méthode `pack`.

Nota 7.4.1. *On utilise l'option `bg` pour changer la couleur de fond et l'option `fg` la couleur du widget.*

7.4.1 Labels

*Un **label** est un espace prévu pour écrire du texte (ou libellé) quelconque (éventuellement une image).*

Les labels servent souvent à décrire un widget comme une entrée.

Exemple 7.4.1.

```
In [1]: from tkinter import *

In [2]: fenetre = Tk()

In [3]: label = Label(label = Label(fenetre, text="Mon premier Label", bg="yellow"))

In [4]: label.pack()

In [5]: fenetre.mainloop()
```

Et on a l'affichage suivant :

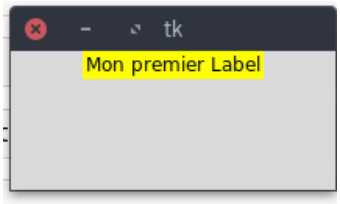

Figure 7.2 – Interface avec un label.

7.4.2 Entry

*Un **Entry** (**Entrée** en français) est une zone de texte dans lequel l'utilisateur peut écrire. En fait de zone, il s'agit d'une ligne simple.*

Exemple 7.4.2.

```
In [1]: from tkinter import *

In [2]: fenetre = Tk()

In [3]: nom = StringVar()

In [4]: nom.set("Veuillez entrer votre nom S.V.P!")

In [5]: entree = Entry(fenetre, textvariable=nom, width=30)

In [6]: entree.pack()

In [7]: fenetre.mainloop()
```

Et on a l'affichage suivant :

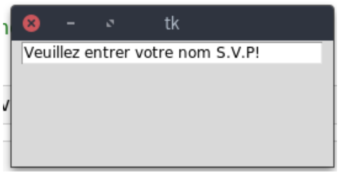

Figure 7.3 – Interface avec une zone d'entrée.

7.4.3 Boutons

*Un **bouton** est un widget permettant de provoquer une action à l'utilisateur. En d'autres termes, il permet de provoquer l'exécution d'une commande quelconque.*

Exemple 7.4.3.

```
In  [10]: from tkinter import *

In  [11]: fenetre = Tk()

In  [12]: boutonQuitter=Button(fenetre, text="Fermer", command=fenetre.quit)

In  [13]: boutonQuitter.pack()

In  [14]: fenetre.mainloop()
```

Et on a l'affichage suivant :

Figure 7.4 – Interface avec un bouton.

7.4.4 Case à cocher

*Une **case à cocher** ou **checkbox** est un widget proposant à l'utilisateur de cocher une option.*

Un clic sur ce widget provoque le changement d'état (la case est cochée ou non).

Pour surveiller l'état d'une case à cocher (qui peut être soit active soit inactive), on préférera créer une variable de type `IntVar` plutôt que `StringVar`, bien que ce ne soit pas une obligation.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXxx

7.4.5 Boutons radio

*Un **bouton radio** est une case à cocher qui est dans un groupe et dans ce groupe seul un élément peut être sélectionné.*

Cependant, cliquer sur un bouton radio donne la valeur correspondante à la variable, et «vide» tous les autres boutons radio associés à la même variable.

Exemple 7.4.4.

```
In [1]: from tkinter import *

In [2]: fenetre = Tk()

In [3]: valeur = StringVar()

In [4]: print("Veuillez choisir un langage de programmation que vous voulez apprendre"
    )
Veuillez choisir un langage de programmation que vous voulez apprendre

In [5]: casePython = Radiobutton(fenetre, text="Langage Python", variable=valeur,
    valeur=1)

In [6]: casePython = Radiobutton(fenetre, text="Langage Python", variable=valeur)

In [7]: casePython = Radiobutton(fenetre, text="Langage Python", variable=valeur,
    value=1)

In [8]: caseC = Radiobutton(fenetre, text="Langage C", variable=valeur, value=2)

In [9]: caseR = Radiobutton(fenetre, text="Langage R", variable=valeur, value=3)

In [10]: caseJava = Radiobutton(fenetre, text="Langage Java", variable=valeur, value
    =4)

In [11]: caseVB = Radiobutton(fenetre, text="Langage VB", variable=valeur, value=5)

In [12]: casePerl = Radiobutton(fenetre, text="Langage Perl", variable=valeur, value
    =6)

In [13]: casePython.pack()

In [14]: caseR.pack()

In [15]: caseC.pack()

In [16]: caseJava.pack()

In [17]: caseVB.pack()

In [18]: casePerl.pack()

In [19]: fenetre.mainloop()
```

Et on a l'affichage suivant :

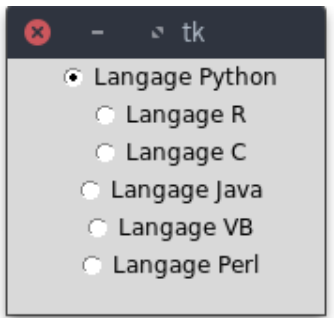

Figure 7.5 – Interface avec des boutons radio.

7.4.6 Listes

*Une **liste de choix** ou **Listbox** est un widget permettant de récupérer une valeur sélectionnée par l'utilisateur.*

On peut également configurer la Listbox de telle manière qu'elle se comporte comme une série de «boutons radio» ou de cases à cocher.

Exemple 7.4.5.

```
In [10]: from tkinter import *

In [11]: fenetre = Tk()

In [12]: liste = Listbox(fenetre)

In [13]: liste.insert(1, "Langage Python")

In [14]: liste.insert(2, "Langage C")

In [15]: liste.insert(3, "Langage R")

In [16]: liste.insert(4, "Langage Java")

In [17]: liste.insert(5, "Langage VB")

In [18]: liste.insert(6, "Langage Perl")

In [19]: liste.pack()

In [20]: fenetre.mainloop()
```

Et on a l'affichage suivant :

7.4.7 Canvas

*Un **canvas** (**toile**, **tableau** en français) est un espace dans lequel on peut dessiner ou écrire ce qu'on veut.*

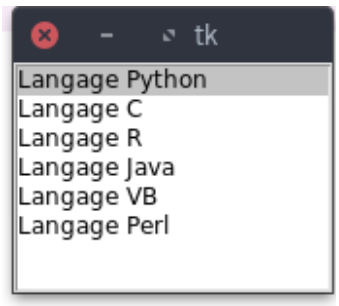

Figure 7.6 – Interface avec une liste.

Ainsi, ce widget peut être utilisé pour dessiner, créer des éditeurs graphiques, et aussi pour implémenter des widgets personnalisés.

7.4.8 Scale

*Une **scale** (**échelle** en francais) est un widget qui permet de récupérer une valeur numérique via un scroll.*

Cependant, il permet de faire varier de manière très visuelle la valeur d'une variable, en déplaçant un curseur le long d'une règle.

7.4.9 Frames

*Les **frames** (**cadres**) sont des conteneurs qui permettent de séparer des éléments.*

Cette surface peut être colorée. Elle peut aussi être décorée d'une bordure.

7.4.10 Labelframe

*Un **labelframe** est un cadre avec un label.*

7.4.11 PanedWindow

*Un **panedwindow** est un conteneur qui peut contenir autant de panneaux que nécessaire disposé horizontalement ou verticalement.*

7.4.12 Spinbox

*Une **spinbox** est un widget qui a pour rôle de proposer à l'utilisateur de choisir un nombre.*

Chapitre 8

PYTHON ET BASES DES DONNÉES RELATIONNELLES

Rappelons qu'une *base de données est un ensemble structuré de données, enregistrées sur des supports, accessibles par l'ordinateur, en vue de satisfaire plusieurs personnes de manière sélective et en temps opportun.*

Cependant, les bases de données qui trouvant dans la quasi-totalité des entreprises aujourd'hui sont les bases des données relationnelle. Les bases de données relationnelle sont des bases de données structurées suivant les principes de l'algèbre relationnelle, créée Edgar Frank Codd.

Le langage standardisé dédié à la structure des bases des données relationnelle est le SQL, qui permet aussi bien de faire des recherches mais aussi des modifications ou des suppressions.

Pour gérer, interroger, manipuler les données ou contrôler les accès aux données, on utilise des logiciels spécifiques qu'on appelle Systèmes de gestion de base de données (SGBD). Dans le cadre de ce chapitre, nous allons apprendre comment accéder aux bases des données relationnelles en utilisant le langage Python. Nous allons donc nous appesantir sur les SGBR SQLite et le MySQL.

8.1 Python et SQLite

SQLite est une bibliothèque écrite en langage C qui propose un moteur de base des données relationnelle accessible par le langage SQL. SQLite a été conçu pour être intégré dans le programme même. Pour faire des projets plus grandes, on utilise le MySQL.

8.1.1 Utilisation de SQLite

Pour utiliser la bibliothèque SQLite, il suffit de l'importer de la manière suivante :

```
In [1]: import sqlite3
```

8.1.2 Création d'une base de données avec SQLite

Pour créer une base des données avec SQLite, il suffit d'utiliser la commande suivant :

```
In [1]: conn = mysql.connector.connect(host="localhost",user="root",password="XXX",
    database="test1")
        cursor = conn.cursor()
        conn.close()
```

8.2 Python et MySQL

MySQL est un système de gestion de base de données relationnelles qui permet de gérer de performances élevé en lecture. Ce qui fait qu'il est adapté pour des applications de grande envergure, comme des applications web.

8.2.1 Utilisation de MySQL

Contrairement à SQLite, il est nécessaire de l'installer et de le configurer. Ensuite, il faut importer la bibliothèque `mysql.connector`.

```
In [1]: import mysql.connector
```

8.2.2 Création d'une base de données avec MySQL

Pour créer une base des données avec MySQL, il faut d'abord se connecter au serveur MySQL.

```
In [1]: conn = mysql.connector.connect(host="localhost",user="root",password="XXX",
    database="test1")
        cursor = conn.cursor()
        conn.close()
```

Exercices

BIBLIOGRAPHIE

[Dar08] C. Darmangeat, *Algorithmique et Programmation pour non-matheux. Cours complet avec exercices, corrigés et citations philosophiques*, Université Paris Diderot–Paris 7, 2008.

[FP17] P. Fuchs and P. Poulain, *Cours de Python*, UFR Sciences du Vivant, Université Paris Diderot–Paris 7, 2017.

[Kuh13] D. Kuhlman, *A Python Book : Beginning Python, Advanced Python, and Python Exercises*, Open Source MIT, 2013.

[MN] M. Matsumoto and T. Nishimura, *Mersenne Twister : A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Transactions on Modeling and Computer Simulation **8**, no. 1, 3–30.

[Pro13] Prolixe, *Apprenez à programmer en Python*, Open Classrooms, Paris, 2013.

[Swi12] G. Swinnen, *Apprendre à programmer avec Python 3*, Eyrolles, Paris, 2012.

[Zia09] T. Ziadé, *Programmation Python. Conception et optimisation*, Eyrolles, Paris, 2009.