UFR MATHÉMATIQUES
UNIVERSITÉ DE RENNES 1

Programmer en Python

Licence 2 - Mathématiques
Notes de cours

Valérie Monbet

4 septembre 2017

Plan du cours

Semaines 1-2-3 Généralités, premiers pas
Semaines 4-5-6 Contrôle de flux, boucles
Semaines 7-8-9 Fonctions, objets
Semaines 10-11-12 Différents types, graphiques, ...

Table des matières

1 Introduction **1**
- 1.1 Avant-propos . . . 1
- 1.2 Choix d'un langage de programmation . . . 1
- 1.3 Caractéristiques du langage Python . . . 1
- 1.4 La démarche du programmeur . . . 2
- 1.5 Langage machine et langage de programmation . . . 2
- 1.6 Edition du code source . . . 3
- 1.7 Installer Python . . . 4

2 Premiers Pas **5**
- 2.1 Calculer avec Python . . . 5
- 2.2 Données et variables . . . 6
- 2.3 Noms de variables et noms réservés . . . 6
- 2.4 Affectation (ou assignation) . . . 7
- 2.5 Afficher la valeur d'une variable . . . 8
- 2.6 Typage des variables . . . 8
- 2.7 Affectations multiples . . . 9
- 2.8 Opérateurs et expressions . . . 9
- 2.9 Composition . . . 10

3 Contrôle du flux d'exécution **12**
- 3.1 Séquence d'instructions . . . 12
- 3.2 Sélection ou exécution conditionnelle . . . 13
- 3.3 Opérateurs de comparaison . . . 14
- 3.4 Les limites des instructions et des blocs sont définies par la mise en page . . . 15
- 3.5 Instruction composée : en-tête, double point, bloc d'instructions indenté . . . 15

4 Instructions répétitives **17**
- 4.1 Répétitions en boucle - l'instruction while . . . 17
- 4.2 Répétitions en boucle - l'instruction for . . . 20

5 Fonctions **22**
- 5.1 Définir une fonction . . . 22
- 5.2 Fonction simple sans paramètres . . . 23
- 5.3 Fonction avec paramètre . . . 24
- 5.4 Fonction avec plusieurs paramètres . . . 25

5.5 Variables locales, variables globales . . . 26
5.6 Fonctions et procédures . . . 27
5.7 Utilisation des fonctions dans un script . . . 28
5.8 Modules de fonctions . . . 29
5.9 Valeurs par défaut pour les paramètres . . . 30

6 Classes, objets, attributs **34**
6.1 Utilité des classes . . . 34
6.2 Définition d'une classe élémentaire . . . 35
6.3 Attributs (ou variables) d'instance . . . 36
6.4 Passage d'objets comme arguments dans l'appel d'une fonction . . . 40
6.5 Similitude et unicité . . . 40

7 Modules existants : matplotlib, numpy **42**
7.1 numpy . . . 42
7.1.1 Objets de type array . . . 42
7.2 Module matplotlib . . . 45

Références

`http://eric.berthomier.free.fr/IMG/pdf/exos_corriges.pdf`

A ajouter

— Listes, tuples, matrices, ...
— Matplotlib
— Importation, exportation de données
— Box, à retenir

Chapitre 1

Introduction

1.1 Avant-propos

Ce cours est très largement inspiré du livre de Gérard Swinnen "Apprendre à programmer avec Python 3".
Licence Creative Commons « Paternité-Pas d'Utilisation Commerciale-Partage des Conditions Initiales à l'Identique - 2.0 France ».

1.2 Choix d'un langage de programmation

Il existe un très grand nombre de langages de programmation, chacun avec ses avantages et ses inconvénients. Il faut bien en choisir un.

Le langage dominant parmi les langages open-source est sans conteste C/C++. Ce langage s'impose comme une référence absolue, et tout informaticien sérieux doit s'y frotter tôt ou tard. Il est malheureusement très rébarbatif et compliqué, trop proche de la machine. Sa syntaxe est peu lisible et fort contraignante.

Python est un langage portable, dynamique, extensible, gratuit, qui permet (sans l'imposer) une approche modulaire et orientée objet de la programmation. Python est développé depuis 1989 par Guido van Rossum et de nombreux contributeurs bénévoles.

1.3 Caractéristiques du langage Python

Détaillons un peu les principales caractéristiques de Python, plus précisément, du langage et de ses deux implantations actuelles :

- Python est **portable**, non seulement sur les différentes variantes d'Unix, mais aussi sur les OS propriétaires : Mac OS, BeOS, NeXTStep, MS-DOS et les différentes variantes de Windows. Un nouveau compilateur, baptisé JPython, est écrit en Java et génère du bytecode Java.
- Python est **gratuit**, mais on peut l'utiliser sans restriction dans des projets commerciaux.
- Python convient aussi bien à des **scripts** d'une dizaine de lignes qu'à des **projets complexes** de plusieurs dizaines de milliers de lignes.

— La **syntaxe** de Python est **très simple** et, combinée à des types de données évolués (listes, dictionnaires...), conduit à des programmes à la fois très compacts et très lisibles.

1.4 La démarche du programmeur

Le mode de pensée d'un programmeur combine des constructions intellectuelles complexes, similaires à celles qu'accomplissent les mathématiciens, les ingénieurs et les scientifiques. Comme le mathématicien, il utilise des langages formels pour décrire des raisonnements (ou algorithmes). Comme l'ingénieur, il conçoit des dispositifs, il assemble des composants pour réaliser des mécanismes et il évalue leurs performances. Comme le scientifique, il observe le comportement de systèmes complexes, il crée des modèles, il teste des prédictions.

L'activité essentielle d'un programmeur consiste à résoudre des problèmes. Il s'agit-là d'une compétence de haut niveau, qui implique des capacités et des connaissances diverses : être capable de (re)formuler un problème de plusieurs manières différentes, être capable d'imaginer des solutions innovantes et efficaces, être capable d'exprimer ces solutions de manière claire et complète. Comme nous l'avons déjà évoqué plus haut, il s'agira souvent de mettre en lumière les implications concrètes d'une représentation mentale « magique », simpliste ou trop abstraite.

Considérons par exemple une suite de nombres fournis dans le désordre : 47, 19, 23, 15, 21, 36, 5, 12 ... Comment devons-nous nous y prendre pour obtenir d'un ordinateur qu'il les remette dans l'ordre ?
Le souhait « magique » est de n'avoir à effectuer qu'un clic de souris sur un bouton, ou entrer une seule instruction au clavier, pour qu'automatiquement les nombres se mettent en place. Mais le travail du sorcier-programmeur est justement de créer cette « magie ». Pour y arriver, il devra décortiquer tout ce qu'implique pour nous une telle opération de tri (au fait, existe-t-il une méthode unique pour cela, ou bien y en a-t-il plusieurs ?), et en traduire toutes les étapes en une suite d'instructions simples, telles que par exemple : « comparer les deux premiers nombres, les échanger s'ils ne sont pas dans l'ordre souhaité, recommencer avec le deuxième et le troisième, etc., etc., ...».

1.5 Langage machine et langage de programmation

À strictement parler, un ordinateur n'est rien d'autre qu'une machine effectuant des opérations simples sur des séquences de signaux électriques, lesquels sont conditionnés de manière à ne pouvoir prendre que deux états seulement (par exemple un potentiel électrique maximum ou minimum). Ces séquences de signaux obéissent à une logique du type « tout ou rien » et peuvent donc être considérés conventionnellement comme des suites de nombres ne prenant jamais que les deux valeurs 0 et 1. Un système numérique ainsi limité à deux chiffres est appelé système binaire.

Sachez dès à présent que dans son fonctionnement interne, un ordinateur est totalement incapable de traiter autre chose que des nombres binaires. Toute information d'un autre type doit être convertie, ou codée, en format binaire. Cela est vrai non seulement pour les données que l'on souhaite traiter (les textes, les images, les sons, les nombres, etc.), mais aussi pour les programmes, c'est-à-dire les séquences d'instructions que l'on va fournir à la machine pour lui dire ce qu'elle doit faire avec ces données.

Le seul « langage » que l'ordinateur puisse véritablement « comprendre » est donc très éloigné de ce que nous utilisons nous-mêmes. C'est une longue suite de 1 et de 0 (les « bits ») souvent traités par groupes de 8 (les « octets »), 16, 32, ou même 64. Ce « langage machine » est évidemment presque incompréhensible pour nous. Pour « parler » à un ordinateur, il nous faudra utiliser des systèmes de traduction automatiques, capables de convertir en nombres binaires des suites de caractères formant des mots-clés (anglais en général) qui seront plus significatifs pour nous.

Le système de traduction proprement dit s'appellera interpréteur ou bien compilateur, suivant la méthode utilisée pour effectuer la traduction. On appellera langage de programmation un ensemble de mots-clés (choisis arbitrairement) associé à un ensemble de règles très précises indiquant comment on peut assembler ces mots pour former des « phrases » que l'interpréteur ou le compilateur puisse traduire en langage machine (binaire).

Python est un langage de haut niveau, dont la traduction en code binaire est complexe et prend donc toujours un certain temps. Cela pourrait paraître un inconvénient. En fait, les avantages que présentent les langages de haut niveau sont énormes : il est beaucoup plus facile d'écrire un programme dans un langage de haut niveau ; l'écriture du programme prend donc beaucoup moins de temps ; la probabilité d'y faire des fautes est nettement plus faible ; la maintenance (c'est-à-dire l'apport de modifications ultérieures) et la recherche des erreurs (les « bugs ») sont grandement facilitées. De plus, un programme écrit dans un langage de haut niveau sera souvent portable, c'est-à-dire que l'on pourra le faire fonctionner sans guère de modifications sur des machines ou des systèmes d'exploitation différents.

1.6 Edition du code source

Le programme tel que nous l'écrivons dans un langage de programmation quelconque est à strictement parler un simple texte (sans mise en forme).

Le texte ainsi produit est ce que nous appellerons désormais un « code source ».

Comme nous l'avons déjà évoqué plus haut, le code source doit être traduit en une suite d'instructions binaires directement compréhensibles par la machine : le « code objet ». Dans le cas de Python, cette traduction est prise en charge par un interpréteur assisté d'un pré-compilateur. Cette technique hybride (également utilisée par le langage Java) vise à exploiter au maximum les avantages de l'interprétation et de la compilation, tout en minimisant leurs inconvénients respectifs.

1.7 Installer Python

Pour installer Python vous pouvez vous référer, par exemple, au site suivant : `http://matthieu-moy.fr/cours/infocpp-S2/install-python.html`

Installation de Python

Installer Python sous Windows

Nous vous recommandons WinPython. Choisir impérativement une version 3.x, par exemple : WinPython 64bit 3.3.5.5.exe.

WinPython peut être utilisé sans être completement installé (par exemple, on peut le mettre sur une clé USB), mais il est recommandé d'« enregistrer » l'installation : voir le bas de cette page pour les explications. Une fois installé, on peut lancer Spyder depuis l'explorateur de fichiers Windows avec un clic droit sur un fichier Python, « Ouvrir avec Spyder ». On trouve aussi Spyder dans le menu applications (menu démarrer/windows).

Spyder, numpy, scipy et matplotlib sont inclus dans WinPython, vous n'avez rien d'autre à installer.

Installer Python sous Mac OS

La solution recommandée est d'utiliser Anaconda : http://continuum.io/downloads (bien sélectionner la version python 3.4)

- Installer anaconda
- Ouvrir le répertoire utilisateur avec le Finder (celui dont l'icone est une maison), et ouvrir le répertoire "anaconda", puis exécuter l'application "Launcher" qui s'y trouve.
- Depuis le Launcher, installer spyder-app, puis l'exécuter.

Installer Python sous Linux

Python et spyder sont inclus dans toute bonne distribution Linux. Attention à choisir une version 3. Sous Ubuntu, installer le paquet python3-spyderlib.

Chapitre 2

Premiers Pas

La programmation est donc l'art de commander à un ordinateur de faire exactement ce que vous voulez, et Python compte parmi les langages qu'il est capable de comprendre pour recevoir vos ordres. Nous allons essayer cela tout de suite avec des ordres très simples concernant des nombres, puisque ce sont les nombres qui constituent son matériau de prédilection. Nous allons lui fournir nos premières « instructions », et préciser au passage la définition de quelques termes essentiels du vocabulaire informatique, que vous rencontrerez constamment dans la suite de cet ouvrage.

2.1 Calculer avec Python

Python présente la particularité de pouvoir être utilisé de plusieurs manières différentes. Vous allez d'abord l'utiliser en mode interactif, c'est-à-dire d'une manière telle que vous pourrez dialoguer avec lui directement depuis le clavier. Cela vous permettra de découvrir très vite un grand nombre de fonctionnalités du langage. Dans un second temps, vous apprendrez comment créer vos premiers programmes (scripts) et les sauvegarder sur disque.

Par exemple, vous pouvez tout de suite utiliser l'interpréteur comme une simple calculatrice de bureau.

```
>>> 5+3
>>> 2 ? 9 # les espaces sont optionnels
>>> 7 + 3 * 4 # la hiérarchie des opérations mathématiques
# est-elle respectée ?
>>> (7+3)*4
>>> 20 / 3 # attention : ceci fonctionnerait différemment sous Python 2
>>> 20 // 3
```

Comme vous pouvez le constater, les opérateurs arithmétiques pour l'addition, la soustraction, la multiplication et la division sont respectivement +, -, * et /. Les parenthèses ont la fonction attendue.

2.2 Données et variables

L'essentiel du travail effectué par un programme d'ordinateur consiste à manipuler des données. Ces données peuvent être très diverses (tout ce qui est numérisable, en fait (6)), mais dans la mémoire de l'ordinateur elles se ramènent toujours en définitive à une suite finie de nombres binaires.

Une variable apparaît dans un langage de programmation sous un `nomdevariable` à peu près quelconque (voir ci après), mais pour l'ordinateur il s'agit d'une référence désignant une adresse mémoire, c'est-à-dire un emplacement précis dans la mémoire vive.

À cet emplacement est stockée une valeur bien déterminée. C'est la donnée proprement dite, qui est donc stockée sous la forme d'une suite de nombres binaires, mais qui n'est pas nécessairement un nombre aux yeux du langage de programmation utilisé. Cela peut être en fait à peu près n'importe quel « objet » susceptible d'être placé dans la mémoire d'un ordinateur, par exemple : un nombre entier, un nombre réel, un nombre complexe, un vecteur, une chaîne de caractères typographiques, un tableau, une fonction, etc.

2.3 Noms de variables et noms réservés

Les noms de variables sont des noms que vous choisissez vous-même assez librement. Efforcez-vous cependant de bien les choisir : de préférence assez courts, mais aussi explicites que possible, de manière à exprimer clairement ce que la variable est censée contenir. Par exemple, des noms de variables tels que altitude, `altit` ou `alt` conviennent mieux que `x` pour exprimer une altitude.

Un bon programmeur doit veiller à ce que ses lignes d'instructions soient faciles à lire.

Sous Python, les noms de variables doivent en outre obéir à quelques règles simples :

— Un nom de variable est une séquence de lettres (a $\rightarrow$ z , A $\rightarrow$ Z) et de chiffres (0 $\rightarrow$ 9), qui doit toujours commencer par une lettre.

— Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que `$, #, @`, etc. sont interdits, à l'exception du caractère `_` (souligné).

— La casse est significative (les caractères majuscules et minuscules sont distingués). Attention : Joseph, joseph, JOSEPH sont donc des variables différentes. Soyez attentifs !

Prenez l'habitude d'écrire l'essentiel des noms de variables en caractères minuscules (y compris la première lettre). Il s'agit d'une simple convention, mais elle est largement respectée. N'utilisez les majuscules qu'à l'intérieur même du nom, pour en augmenter éventuellement la lisibilité, comme dans `tableDesMatieres`.

En plus de ces règles, il faut encore ajouter que vous ne pouvez pas utiliser comme nom de variables les 33 « mots réservés » ci-dessous (ils sont utilisés par le langage lui-même) :

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

2.4 Affectation (ou assignation)

En Python comme dans de nombreux autres langages, l'opération d'affectation est représentée par le signe =.

```
>>> n = 7 # définir n et lui donner la valeur 7
>>> msg = "Quoi de neuf ?" # affecter la valeur "Quoi de neuf ?" à msg
>>> pi = 3.14159 # assigner sa valeur à la variable pi
```

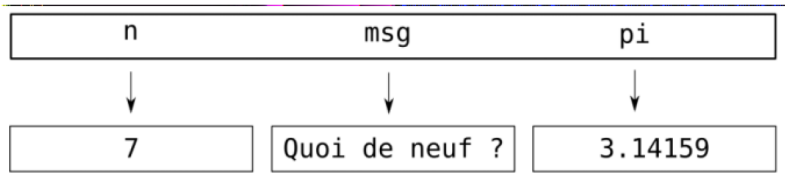
Les exemples ci-dessus illustrent des instructions d'affectation Python tout à fait classiques. Après qu'on les ait exécutées, il existe dans la mémoire de l'ordinateur, à des endroits différents :

— trois noms de variables, à savoir `n`, `msg` et `pi` ;
— trois séquences d'octets, où sont encodées le nombre entier 7, la chaîne de caractères `Quoi de neuf ?` et le nombre réel 3,14159.

Les trois instructions d'affectation ci-dessus ont eu pour effet chacune de réaliser plusieurs opérations dans la mémoire de l'ordinateur :

— créer et mémoriser un nom de variable ;
— lui attribuer un type bien déterminé (ce point sera explicité à la page suivante) ;
— créer et mémoriser une valeur particulière ;
— établir un lien (par un système interne de pointeurs) entre le nom de la variable et l'emplacement mémoire de la valeur correspondante.

On peut mieux se représenter tout cela par un diagramme d'état tel que celui-ci :



Les trois noms de variables sont des *références*, mémorisées dans une zone particulière de la mémoire que l'on appelle espace de noms, alors que les valeurs correspondantes sont situées ailleurs, dans des emplacements parfois fort éloignés les uns des autres.

On peut aussi affecter des listes (de chiffres ou de chaînes de caractères) à une variable et accéder au $k$ième élément de la liste de la façon suivante.

```
>>> premiers = [1,3,5,7,11]
>>> premiers[1]
3
>>> premiers[4]
11
>>> premiers[0]+premiers[1]
4
```

Vous remarquerez que l'indexation commence à 0. En effet

```
>>> premiers = [0]
1
```

Un autre exemple

```
>>> marque = ["citroen","peugeot","renault"]
>>> print(marque)
['citroen', 'peugeot', 'renault']
>>> modele = ["berlingot","partner","kangoo"]
>>> modele[1]
'partner'
```

2.5 Afficher la valeur d'une variable

À la suite de l'exercice ci-dessus, nous disposons donc des trois variables `n`, `msg` et `pi`. Pour afficher leur valeur à l'écran, il existe deux possibilités. La première consiste à entrer au clavier le nom de la variable, puis `<Enter>`. Python répond en affichant la valeur correspondante :

```
>>> n
7
>>> msg
'Quoi de neuf ?'
>>> pi
3.14159
```

Il s'agit cependant là d'une fonctionnalité secondaire de l'interpréteur, qui est destinée à vous faciliter la vie lorsque vous faites de simples exercices à la ligne de commande. À l'intérieur d'un programme, vous utiliserez toujours la fonction `print()`

```
>>> print(msg)
Quoi de neuf ?
>>> print(n)
7
```

2.6 Typage des variables

Sous Python, il n'est pas nécessaire d'écrire des lignes de programme spécifiques pour définir le type des variables avant de pouvoir les utiliser. Il vous suffit en effet d'assigner une valeur à un nom de variable pour que celle-ci soit automatiquement créée avec le type qui correspond au mieux à la valeur fournie. Dans l'exercice précédent, par exemple, les variables `n`, `msg` et `pi` ont été créées automatiquement chacune avec un type différent (« nombre entier » pour `n`, « chaîne de caractères » pour `msg`, « nombre à virgule flottante » (ou « float », en anglais) pour `pi`).

On dira à ce sujet que le typage des variables sous Python est un *typage dynamique*, par opposition au *typage statique* qui est de règle par exemple en C++ ou en Java.

2.7 Affectations multiples

Sous Python, on peut assigner une valeur à plusieurs variables simultanément. Exemple :

```
>>> x = y = 7
>>> print(x)
7
>>> print(y)
7
```

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

```
>>> a, b = 4, 8.33
>>> a
4
>>> b
8.33
```

Dans cet exemple, les variables `a` et `b` prennent simultanément les nouvelles valeurs 4 et 8.33.

Exercices

1. *Décrivez le plus clairement et le plus complètement possible ce qui se passe à chacune des trois lignes de l'exemple ci-dessous :*

   ```
   >>> largeur = 20
   >>> hauteur = 5 * 9.3
   >>> largeur * hauteur
   ```

2. *Assignez les valeurs respectives 3, 5, 7 à trois variables a, b, c. Effectuez l'opération a - b//c. Interprétez le résultat obtenu.*

2.8 Opérateurs et expressions

On manipule les valeurs et les variables qui les référencent en les combinant avec des *opérateurs* pour former des expressions. Exemple :

```
a, b = 7.3, 12
y = 3*a + b/5
```

Exercice
Dans les commandes ci-dessus quel est le type affecté automatiquement à la variable `a` *? à la variable* `b` *?*

Python évalue chaque expression qu'on lui soumet, aussi compliquée soit-elle, et le résultat de cette évaluation est toujours lui-même une valeur. À cette valeur, il attribue automatiquement un type, lequel dépend de ce qu'il y a dans l'expression. Dans l'exemple ci-dessus, y sera du type réel, parce que l'expression évaluée pour déterminer sa valeur contient elle-même au moins un

réel. Les opérateurs Python ne sont pas seulement les quatre opérateurs mathématiques de base. Nous avons déjà signalé l'existence de l'opérateur de division entière `//`. Il faut encore ajouter l'opérateur `**` pour l'exponentiation, ainsi qu'un certain nombre d'opérateurs logiques, des opérateurs agissant sur les chaînes de caractères, des opérateurs effectuant des tests d'identité ou d'appartenance, etc. Nous reparlerons de tout cela plus loin.

Signalons au passage la disponibilité de l'opérateur modulo, représenté par le caractère typographique opérateur fournit le reste de la division entière d'un nombre par un autre. Par exemple :

```
>>> 10 % 3 # (et prenez note de ce qui se passe !)
>>> 10 % 5
```

Exercice
Testez les lignes d'instructions suivantes. Décrivez dans votre cahier ce qui se passe :

```
>>> r , pi = 12, 3.14159
>>> s = pi * r**2
>>> print(s)
>>> print(type(r), type(pi), type(s))
```

Quelle est, à votre avis, l'utilité de la fonction `type()` *?*

2.9 Composition

Jusqu'ici nous avons examiné les différents éléments d'un langage de programmation, à savoir : les *variables*, les *expressions* et les *instructions*, mais sans traiter de la manière dont nous pouvons les combiner les unes avec les autres.

Or l'une des grandes forces d'un langage de programmation de haut niveau est qu'il permet de construire des instructions complexes par assemblage de fragments divers. Ainsi par exemple, si vous savez comment additionner deux nombres et comment afficher une valeur, vous pouvez combiner ces deux instructions en une seule :

```
>>> print(17 + 3)
>>> 20
```

Cela n'a l'air de rien, mais cette fonctionnalité qui paraît si évidente va vous permettre de programmer des algorithmes complexes de façon claire et concise. Exemple :

```
>>> h, m, s = 15, 27, 34
>>> print("nombre de secondes écoulées depuis minuit = ", h*3600 + m*60 + s)
```

Dans une expression, ce que vous placez à la gauche du signe égale doit toujours être une variable, et non une expression. Cela provient du fait que le signe égale n'a pas ici la même signification qu'en mathématique : comme nous l'avons déjà signalé, il s'agit d'un symbole d'affectation (nous plaçons un certain contenu dans une variable) et non un symbole d'égalité.

A retenir

- L'interpréteur Python peut être utilisé comme calculatrice. Plus généralement, les instructions peuvent être tappées directement dans l'interpréteur.
- La priorité des opérations et les parenthèses suivent les mêmes règles qu'en mathématiques.
- Il est possible de faire des affectations multiples.
- Pour afficher des valeurs ou des chaines de caractères, on utilise la fonction `print`.
- Certains noms de variables sont réservés.

Chapitre 3

Contrôle du flux d'exécution

Dans notre premier chapitre, nous avons vu que l'activité essentielle d'un programmeur est la résolution de problèmes. Or, pour résoudre un problème informatique, il faut toujours effectuer une série d'actions dans un certain ordre. La description structurée de ces actions et de l'ordre dans lequel il convient de les effectuer s'appelle un *algorithme*.

Le « chemin » suivi par Python à travers un programme est appelé un *flux d'exécution*, et les constructions qui le modifient sont appelées des *instructions de contrôle de flux*. Les structures de contrôle sont les groupes d'instructions qui déterminent l'ordre dans lequel les actions sont effectuées. En programmation moderne, il en existe seulement trois : la séquence et la sélection, que nous allons décrire dans ce chapitre, et la répétition que nous aborderons au chapitre suivant.

3.1 Séquence d'instructions

Sauf mention explicite, les instructions d'un programme s'exécutent les unes après les autres, dans l'ordre où elles ont été écrites à l'intérieur du script.

Cette affirmation peut vous paraître banale et évidente à première vue. L'expérience montre cependant qu'un grand nombre d'erreurs sémantiques dans les programmes d'ordinateur sont la conséquence d'une mauvaise disposition des instructions. Plus vous progresserez dans l'art de la programmation, plus vous vous rendrez compte qu'il faut être extrêmement attentif à l'ordre dans lequel vous placez vos instructions les unes derrière les autres. Par exemple, dans la séquence d'instructions suivantes :

```
>>> a, b = 3, 7
>>> a = b
>>> b = a
>>> print(a, b)
```

Vous obtiendrez un résultat contraire si vous intervertissez les 2e et 3e lignes.

Python exécute normalement les instructions de la première à la dernière, sauf lorsqu'il rencontre une instruction conditionnelle comme l'instruction `if`. Une telle instruction va permettre au programme de suivre différents chemins suivant les circonstances.

3.2 Sélection ou exécution conditionnelle

Si nous voulons pouvoir écrire des applications véritablement utiles, il nous faut des techniques permettant d'aiguiller le déroulement du programme dans différentes directions, en fonction des circonstances rencontrées. Pour ce faire, nous devons disposer d'instructions capables de tester une certaine condition et de modifier le comportement du programme en conséquence.

La plus simple de ces instructions conditionnelles est l'instruction `if`.

```
>>> a = 150
>>> if (a > 100):
...     print("a dépasse la centaine")
...
```

Si vous tapez sur la touche *Entrée*, le programme s'exécute, et vous obtenez :

```
a dépasse la centaine
```

Recommencez le même exercice, mais avec a = 20 en guise de première ligne : cette fois Python n'affiche plus rien.

L'expression que vous avez placée entre parenthèses est ce que nous appellerons désormais une *condition*. L'expression que vous avez placée entre parenthèses est ce que nous appellerons désormais 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. Si la condition est fausse, rien ne se passe.

```
>>> a = 20
>>> if (a > 100):
...     print("a dépasse la centaine")
...else:
...     print("a ne dépasse pas cent")
...
```

l'instruction `else` ("sinon", en anglais) permet de programmer une exécution alternative, dans laquelle le programme doit choisir entre deux possibilités. On peut faire mieux encore en utilisant aussi l'instruction `elif` (contraction de "else if") :

```
>>> a = 0
>>> if a > 0 :
...     print("a est positif")
... elif a < 0 :
...     print("a est négatif")
... else:
...     print("a est nul")
...
```

3.3 Opérateurs de comparaison

La condition évaluée après l'instruction if peut contenir les *opérateurs de comparaison* suivants (à connaitre ! ! !) :

```
x == y   # x est égal à y
x != y    # x est différent de y
x > y     # x est plus grand que y
x < y     # x est plus petit que y
x >= y   # x est plus grand que, ou égal à y
x <= y   # x est plus petit que, ou égal à y
```

Exercices

1. *Écrire un programme qui affiche "vous êtes majeur!" si votre âge est supérieur ou égale à 18 et "vous êtes mineur." sinon.*
2. *Écrire un programme qui affiche "a est pair" si la valeur de la variable* `a` *est paire et "a est impair" sinon. Vous utiliserez l'opérateur modulo dans la condition.*
3. *Dans quel cas le programme suivant affiche t'il "c'est peut-être un chat" ?*

```
if embranchement == "vertébrés": # 1
   if classe == "mammifères": # 2
      if ordre == "carnivores": # 3
      if famille == "félins": # 4
   print("c'est peut-être un chat") # 5
      print("c'est en tous cas un mammifère") # 6
   elif classe == "oiseaux": # 7
      print("c'est peut-être un canari") # 8
print("la classification des animaux est complexe") # 9
```

4. *Qu'affiche le programme ci-dessus si on affecte*

```
embranchement, classe = "vertébrés", "oiseaux"
```

   avant de l'exécuter.

5. *Écrire et tester un programme* `maximum.py` *qui demande á l'utilisateur deux nombres, puis affiche la phrase Le nombre $\cdots$ est certainement plus grand que $\cdots$, en remplissant correctement les pointillés.*
6. *Modifier le programme précédent en prévoyant d'afficher éventuellement la phrase Les deux nombres sont égaux, lorsque les deux nombres saisis seront égaux.*

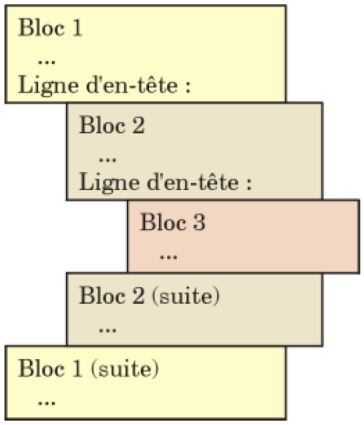
3.4 Les limites des instructions et des blocs sont définies par la mise en page

Dans de nombreux langages de programmation, il faut terminer chaque ligne d'instructions par un caractère spécial (souvent le point-virgule). Sous Python, c'est le caractère de fin de ligne qui joue ce rôle. (Nous verrons plus loin comment outrepasser cette règle pour étendre une instruction complexe sur plusieurs lignes.) On peut également terminer une ligne d'instructions par un commentaire. Un commentaire Python commence toujours par le caractère spécial #. Tout ce qui est inclus entre ce caractère et le saut à la ligne suivant est complètement ignoré par le compilateur.

Avec Python, vous **devez** utiliser les sauts à la ligne et l'indentation, mais en contrepartie vous n'avez pas à vous préoccuper d'autres symboles délimiteurs de blocs. En définitive, Python vous force donc à écrire du code lisible, et à prendre de bonnes habitudes que vous conserverez lorsque vous utiliserez d'autres langages.

3.5 Instruction composée : en-tête, double point, bloc d'instructions indenté

Le schéma ci-contre résume le principe des blocs d'instructions imbriqués.



— Les blocs d'instructions sont toujours associés à une ligne d'en-tête contenant une instruction bien spécifique (if, elif, else, while, def, etc.) se terminant par un double point.
— Les blocs sont délimités par l'indentation : toutes les lignes d'un même bloc doivent être indentées exactement de la même manière (c'est-à-dire décalées vers la droite d'un même nombre d'espaces). Le nombre d'espaces à utiliser pour l'indentation est quelconque, mais la plupart des programmeurs utilisent des multiples de 4.
— Notez que le code du bloc le plus externe (bloc 1) ne peut pas lui-même être écarté de la marge de gauche (il n'est imbriqué dans rien).

Exercices

1. *On désire sécuriser une enceinte pressurisée. On se fixe une pression seuil et un volume seuil :* `pSeuil = 2.3`, `vSeuil = 7.41`. *On demande de saisir la pression et le volume courant de l'enceinte et d'écrire un script qui simule le comportement suivant :*
   - *si le volume et la pression sont supérieurs aux seuils : arrêt immédiat ;*
   - *si seule la pression est supérieure à la pression seuil : demander d'augmenter le volume de l'enceinte ;*
   - *si seul le volume est supérieur au volume seuil : demander de diminuer le volume de l'enceinte ;*
   - *sinon déclarer que « tout va bien ».*

   Ce comportement sera implémenté par une alternative multiple.

A retenir

- L'exécution conditionnelle d'instructions est réalisée en utilisant le `if` ;
  ou le `if` $\cdots$ `else` ;
  ou le `if` $\cdots$ `elif` $\cdots$ `else`.
- Les blocs d'instructions sont toujours associés à une ligne d'en-tête contenant une instruction bien spécifique (if, elif, else, while, def, etc.) se terminant par un double point `:`.
- Les blocs sont délimités par l'indentation : toutes les lignes d'un même bloc doivent être indentées exactement de la même manière.

Chapitre 4

Instructions répétitives

L'une des tâches que les machines font le mieux est la répétition sans erreur de tâches identiques. Il existe bien des méthodes pour programmer ces tâches répétitives. Nous allons commencer par l'une des plus fondamentales : la boucle de répétition construite autour de l'instruction `while`.

4.1 Répétitions en boucle - l'instruction while

Un premier exemple.

```
>>> a = 0
>>> while (a < 7): # (n'oubliez pas le double point !)
...     a = a + 1 # (n'oubliez pas l'indentation !)
...     print(a)
```

Le mot while signifie "tant que" en anglais. Cette instruction utilisée à la seconde ligne indique à Python qu'il lui faut répéter continuellement le bloc d'instructions qui suit, tant que le contenu de la variable a reste inférieur à 7.

Comme l'instruction `if` abordée au chapitre précédent, l'instruction while amorce une instruction composée. Le double point à la fin de la ligne introduit le bloc d'instructions à répéter, lequel doit obligatoirement se trouver en retrait. Comme vous l'avez appris au chapitre précédent, toutes les instructions d'un même bloc doivent être indentées exactement au même niveau (c'est-à-dire décalées à droite d'un même nombre d'espaces).

Nous avons ainsi construit notre première boucle de programmation, laquelle répète un certain nombre de fois le bloc d'instructions indentées. Voici comment cela fonctionne :

— Avec l'instruction while, Python commence par évaluer la validité de la condition fournie entre parenthèses (celles-ci sont optionnelles, nous ne les avons utilisées que pour clarifier notre explication).
— Si la condition se révèle fausse, alors tout le bloc qui suit est ignoré et l'exécution du programme se termine.
— Si la condition est vraie, alors Python exécute tout le bloc d'instructions constituant le corps de la boucle, c'est-à-dire :

    - l'instruction a = a + 1 qui incrémente d'une unité le contenu de la variable a (ce qui signifie que l'on affecte à la variable a une nouvelle valeur, qui est égale à la valeur précédente augmentée d'une unité).
    - l'appel de la fonction `print()` pour afficher la valeur courante de la variable a.
- lorsque ces deux instructions ont été exécutées, nous avons assisté à une première itération, et le programme boucle, c'est-à-dire que l'exécution reprend à la ligne contenant l'instruction while. La condition qui s'y trouve est à nouveau évaluée, et ainsi de suite. Dans notre exemple, si la condition `a < 7` est encore vraie, le corps de la boucle est exécuté une nouvelle fois et le bouclage se poursuit.
- La variable évaluée dans la condition doit exister au préalable (il faut qu'on lui ait déjà affecté au moins une valeur).
- Si la condition est fausse au départ, le corps de la boucle n'est jamais exécuté.
- Si la condition reste toujours vraie, alors le corps de la boucle est répété indéfiniment (tout au moins tant que Python lui-même continue à fonctionner). Il faut donc veiller à ce que le corps de la boucle contienne au moins une instruction qui change la valeur d'une variable intervenant dans la condition évaluée par while, de manière à ce que cette condition puisse devenir fausse et la boucle se terminer.

Exemple : élaboration de tables

```
>>> a = 0
>>> while (a < 7): # (n'oubliez pas le double point !)
...     a = a + 1 # (n'oubliez pas l'indentation !)
...     print(a)
```

Exemple : construction d'une suite mathématique

```
>>> a, b, c = 1, 1, 1
>>> while c < 11 :
... print(b, end =" ")
... a, b, c = b, a+b, c+1
```

Les termes de la suite de Fibonacci sont affichés sur la même ligne. Vous obtenez ce résultat grâce au second argument `end =" "` fourni à la fonction `print()`.

Exercices

1. *Écrivez un programme qui affiche les 20 premiers termes de la table de multiplication par 7.*
2. *Écrivez un programme qui affiche une table de conversion de sommes d'argent exprimées en euros, en dollars canadiens. La progression des sommes de la table sera « géométrique », comme dans l'exemple ci-dessous :*

   ```
   1 euro(s) = 1.65 dollar(s)
   2 euro(s) = 3.30 dollar(s)
   4 euro(s) = 6.60 dollar(s)
   8 euro(s) = 13.20 dollar(s)
   ```

   S'arrêter à 16384 euros.
3. *Pour chacun des programmes suivant, prédire le résultat*

   (a)
   ```
   cpt = 0
   while (cpt < 100):
       somme = cpt
       cpt = cpt + 1
   print somme
   ```

   (b)
   ```
   cpt = 0
   somme = 0
   while (cpt < 100):
       somme = cpt
       cpt = cpt + 1
   print somme
   ```

   (c)
   ```
   cpt = 0
   somme = 0
   while (cpt < 100):
       produit = somme * cpt
       print produit
       cpt = somme + 1
   print somme
   ```

   (d)
   ```
   cpt = 0
   somme = 0
   while (cpt < 10):
   cpt = somme - 1
   cpt = cpt + 1
   print somme
   ```
4. *Écrire un programme qui affiche tous les nombres impairs entre 0 et 150, dans l'ordre croissant.*
5. *Écrire un programme qui affiche tous les nombres impairs entre 0 et 150, dans l'ordre décroissant.*

4.2 Répétitions en boucle - l'instruction for

On peut aussi répéter des instructions en utilisant les boucles `for`. Exemple

```
>>> a, b = 1,1
>>> for c in range(1,11) :
... print(b, end =" ")
... a, b = b, a+b
```

Dans ce cas, la variable `c` est indentée par l'instruction `for`.

La fonction `range()` permet de créer une liste d'entiers. Elle s'utilise avec la syntaxe suivante `range(start,stop[step],)` où `start` correspond à l'entier de départ, `stop` à l'entier final et `step` est optionnel et permet de faire des pas différents de 1.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
>>> range(1, 0)
[]
```

On note que par défaut, la liste comment à 0.

On peut aussi manipuler facilement des chaînes de caractères avec les boucles `for`.

```
>>> v = "Bonjour toi"
>>> for lettre in v:
...     print lettre
```

L'instruction `break` permet de stopper la boucle quand on rencontre une condition.

```
>>> liste = [1,5,10,15,20,25]
>>> for i in liste:
...     if i > 15:
...             print "On stoppe la boucle"
...             break
...     print i
...
```

Exercices

1. *Initialisez deux entiers : a = 0 et b = 10 . Écrire une boucle affichant et incrémentant la valeur de a tant qu'elle reste inférieure à celle de b . Écrire une autre boucle décrémentant la valeur de b et affichant sa valeur si elle est impaire. Boucler tant que b n'est pas nul.*
2. *Affichez les entiers de 0 à 15 non compris, de trois en trois, en utilisant une boucle* `for` *et l'instruction* `range()` .
3. *Écrivez un programme qui calcule le volume d'un parallélépipède rectangle dont sont fournis au départ la largeur, la hauteur et la profondeur.*
4. *Écrivez un programme qui convertit un nombre entier de secondes fourni au départ en un nombre d'années, de mois, de jours, de minutes et de secondes (utilisez l'opérateur modulo :*
5. *Écrivez un programme qui affiche les 20 premiers termes de la table de multiplication par 7, en signalant au passage (à l'aide d'une astérisque) ceux qui sont des multiples de 3. Exemple : 7 14 21 * 28 35 42 * 49 ...*

A retenir

- Les répétitions en boucle sont réalisées via l'instruction `while` ou l'instruction `for`.
- Le mot clé `while` permet de répéter un bloc d'instructions tant que la condition n'est pas vérifiée.
- Le mot clé `for` permet de répéter un bloc d'instructions un certain nombre de fois commandé par l'énumération du `for`.
- La fonction `range(start,stop,step)` génère une liste débutant à `start`, indentée de `step`à chaque itération et s'arrêtant à `stop-step`.
  Par défaut, `start=0` et `step=1`.
- La commande `break` permet de stopper l'exécution d'un une boucle.

Chapitre 5

Fonctions

La programmation est l'art d'apprendre à un ordinateur comment accomplir des tâches qu'il n'était pas capable de réaliser auparavant. L'une des méthodes les plus intéressantes pour y arriver consiste à ajouter de nouvelles instructions au langage de programmation que vous utilisez, sous la forme de fonctions originales.

5.1 Définir une fonction

Les scripts que vous avez écrits jusqu'à présent étaient à chaque fois très courts, car leur objectif était seulement de vous faire assimiler les premiers éléments du langage. Lorsque vous commencerez à développer de véritables projets, vous serez confrontés à des problèmes souvent fort complexes, et les lignes de programme vont commencer à s'accumuler...

L'approche efficace d'un problème complexe consiste souvent à le décomposer en plusieurs sous-problèmes plus simples qui seront étudiés séparément (ces sous-problèmes peuvent éventuellement être eux-mêmes décomposés à leur tour, et ainsi de suite). Or il est important que cette décomposition soit représentée fidèlement dans les algorithmes pour que ceux-ci restent clairs.

Les *fonctions* et les *classes d'objets* sont différentes structures de sous-programmes qui ont été imaginées par les concepteurs des langages de haut niveau afin de résoudre les difficultés évoquées ci-dessus. Nous allons commencer par décrire ici la *définition de fonctions* sous Python. Les *objets* et les *classes* seront examinés plus loin.

La syntaxe Python pour la définition d'une fonction est la suivante :

```
def nomDeLaFonction(liste de paramètres):
    ...
    bloc d'instructions
    ...
```

— Vous pouvez choisir n'importe quel nom pour la fonction que vous créez, à l'exception des mots réservés du langage, 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 plus loin).

— Comme les instructions `if` et `while` que vous connaissez déjà, l'instruction `def` est une *instruction composée*. La ligne contenant cette instruction se termine obligatoirement par un double point, lequel introduit un bloc d'instructions que vous ne devez pas oublier d'indenter.
— La *liste de paramètres* 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).
— Une fonction s'utilise pratiquement comme une instruction quelconque. Dans le corps d'un programme, un appel de fonction est constitué du nom de la fonction suivi de parenthèses. Si c'est nécessaire, on place dans ces parenthèses le ou les arguments que l'on souhaite transmettre à la fonction. Il faudra en principe fournir un argument pour chacun des paramètres spécifiés dans la définition de la fonction, encore qu'il soit possible de définir pour ces paramètres des valeurs par défaut (voir plus loin).

5.2 Fonction simple sans paramètres

Les quelques lignes ci-dessous définissent une fonction très simple qui calcule et affiche les 10 premiers termes de la table de multiplication par 7. Notez bien les parenthèses, le double point, et l'indentation du bloc d'instructions qui suit la ligne d'en-tête (c'est ce bloc d'instructions qui constitue le corps de la fonction proprement dite).

```
>>> def table7():
...     n = 1
...     while n <11 :
...         print(n * 7, end =' ')
...         n = n +1
```

Pour utiliser la fonction que nous venons de définir, il suffit de l'appeler par son nom. Ainsi :

```
>>> table7()
```

provoque l'affichage de :

```
7 14 21 28 35 42 49 56 63 70
```

Nous pouvons maintenant réutiliser cette fonction à plusieurs reprises, autant de fois que nous le souhaitons. Nous pouvons également l'incorporer dans la définition d'une autre fonction, comme dans l'exemple ci-dessous :

```
>>> def table7triple():
...     print('La table par 7 en triple exemplaire :')
...     table7()
...     table7()
...     table7()
...
```

Utilisons cette nouvelle fonction, en entrant la commande :

```
>>> >>> table7triple()
```

```
La table par 7 en triple exemplaire :
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
```

Une première fonction peut donc appeler une deuxième fonction, qui elle-même en appelle une troisième, etc. Au stade où nous sommes, vous ne voyez peut-être pas encore très bien l'utilité de tout cela, mais vous pouvez déjà noter deux propriétés intéressantes :

— Créer une nouvelle fonction vous offre l'opportunité de donner un nom à tout un ensemble d'instructions. De cette manière, vous pouvez simplifier le corps principal d'un programme, en dissimulant un algorithme secondaire complexe sous une commande unique, à laquelle vous pouvez donner un nom très explicite, en français si vous voulez.

— Créer une nouvelle fonction peut servir à raccourcir un programme, par élimination des portions de code qui se répètent. Par exemple, si vous devez afficher la table par 7 plusieurs fois dans un même programme, vous n'avez pas à réécrire chaque fois l'algorithme qui accomplit ce travail. Une fonction est donc en quelque sorte une nouvelle instruction personnalisée, que vous ajoutez vous-même librement à votre langage de programmation.

Une fonction est donc en quelque sorte une nouvelle instruction personnalisée, que vous ajoutez vous-même librement à votre langage de programmation.

5.3 Fonction avec paramètre

Dans nos derniers exemples, nous avons défini et utilisé une fonction qui affiche les termes de la table de multiplication par 7. Supposons à présent que nous voulions faire de même avec la table par 9. Nous pouvons bien entendu réécrire entièrement une nouvelle fonction pour cela. Mais si nous nous intéressons plus tard à la table par 13, il nous faudra encore recommencer. Ne serait-il donc pas plus intéressant de définir une fonction qui soit capable d'afficher n'importe quelle table, à la demande ?

Lorsque nous appellerons cette fonction, nous devrons bien évidemment pouvoir lui indiquer quelle table nous souhaitons afficher. Cette information que nous voulons transmettre à la fonction au moment même où nous l'appelons s'appelle un *argument*. Nous avons déjà rencontré à plusieurs reprises des fonctions intégrées qui utilisent des arguments. La fonction `sin(a)`, par exemple, calcule le sinus de l'angle `a`. La fonction `sin()` utilise donc la valeur numérique de `a` comme argument pour effectuer son travail.

Dans la définition d'une telle fonction, il faut prévoir une variable particulière pour recevoir l'argument transmis. Cette variable particulière s'appelle un *paramètre*. On lui choisit un nom en respectant les mêmes règles de syntaxe que d'habitude (pas de lettres accentuées, etc.), et on place ce nom entre les parenthèses qui accompagnent la définition de la fonction.

```
>>> def table(base):
...     n = 1
```

```
...     while n <11 :
...         print(n * base, end =' ')
...         n = n +1
```

La fonction `table()` telle que définie ci-dessus utilise le paramètre `base` pour calculer les dix premiers termes de la table de multiplication correspondante. Pour tester cette nouvelle fonction, il nous suffit de l'appeler avec un argument. Exemples :

```
>>> table(13)
13 26 39 52 65 78 91 104 117 130
>>> table(9)
9 18 27 36 45 54 63 72 81 90
```

L'argument que nous utilisons dans l'appel d'une fonction peut être une variable lui aussi, comme dans l'exemple ci-dessous. Par exemple

```
>>> a = 1
>>> while a <20:
...     table(a)
...     a = a +1
...
```

Remarque importante : Le nom d'une variable que nous passons comme argument n'a rien à voir avec le nom du paramètre correspondant dans la fonction.

5.4 Fonction avec plusieurs paramètres

La fonction `table()` est certainement intéressante, mais elle n'affiche toujours que les dix premiers termes de la table de multiplication, alors que nous pourrions souhaiter qu'elle en affiche d'autres. Qu'à cela ne tienne. Nous allons l'améliorer en lui ajoutant des paramètres supplémentaires, dans une nouvelle version que nous appellerons cette fois `tableMulti()` :

```
>>> def tableMulti(base, debut, fin):
...     print('Fragment de la table de multiplication par', base, ':')
...     n = debut
...     while n <= fin :
...         print(n, 'x', base, '=', n * base)
...         n = n +1
```

Cette nouvelle fonction utilise donc trois paramètres : la base de la table comme dans l'exemple précédent, l'indice du premier terme à afficher, l'indice du dernier terme à afficher.

Essayons cette fonction en entrant par exemple :

```
>>> tableMulti(8, 13, 17)
```

ce qui devrait provoquer l'affichage de :

```
Fragment de la table de multiplication par 8 :
13 x 8 = 104
14 x 8 = 112
15 x 8 = 120
16 x 8 = 128
17 x 8 = 136
```

— Pour définir une fonction avec plusieurs paramètres, il suffit d'inclure ceux-ci entre les parenthèses qui suivent le nom de la fonction, en les séparant à l'aide de virgules.
— Lors de l'appel de la fonction, les arguments utilisés doivent être fournis dans le même ordre que celui des paramètres correspondants (en les séparant eux aussi à l'aide de virgules). Le premier argument sera affecté au premier paramètre, le second argument sera affecté au second paramètre, et ainsi de suite.

Exercice
Quel est le résultat obtenu par les instructions suivantes :

```
>>> t, d, f = 11, 5, 10
>>> while t<21:
...     tableMulti(t,d,f)
...     t, d, f = t +1, d +3, f +5
...
```

5.5 Variables locales, variables globales

Lorsque nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des variables locales à la fonction. C'est par exemple le cas des variables `base`, `debut`, `fin` et `n` dans l'exercice précédent. Chaque fois que la fonction `tableMulti()` est appelée, Python réserve pour elle (dans la mémoire de l'ordinateur) un nouvel espace de noms. Les contenus des variables `base`, `debut`, `fin` et `n` sont stockés dans cet espace de noms qui est inaccessible depuis l'extérieur de la fonction. Ainsi par exemple, si nous essayons d'afficher le contenu de la variable base juste après avoir effectué l'exercice ci-dessus, nous obtenons un message d'erreur :

```
>>> print(base)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'base' is not defined
```

La machine nous signale clairement que le symbole base lui est inconnu, alors qu'il était correctement imprimé par la fonction `tableMulti()` elle-même. L'espace de noms qui contient le symbole base est strictement réservé au fonctionnement interne de`tableMulti()`, et il est automatiquement détruit dès que la fonction a terminé son travail.

Les variables définies à l'extérieur d'une fonction sont des variables globales. Leur contenu est « visible » de l'intérieur d'une fonction, mais la fonction ne peut pas le modifier. Exemple :

```
>>> def mask():
... p = 20
... print(p, q)
...
>>> p, q = 15, 38
>>> mask()
20 38
>>> print(p, q)
15 38
```

5.6 Fonctions et procédures

Pour les puristes, les fonctions que nous avons décrites jusqu'à présent ne sont pas tout à fait des fonctions au sens strict, mais plus exactement des *procédures* . Une « vraie » fonction (au sens strict) doit en effet renvoyer une valeur lorsqu'elle se termine. Une « vraie » fonction peut s'utiliser à la droite du signe égale dans des expressions telles que `y = sin(a)`. On comprend aisément que dans cette expression, la fonction `sin()` renvoie une valeur (le sinus de l'argument `a`) qui est directement affectée à la variable `y`.

Commençons par un exemple extrêmement simple :

```
>>> def cube(w):
...     return w*w*w
...
```

L'instruction return définit ce que doit être la valeur renvoyée par la fonction. En l'occurrence, il s'agit du cube de l'argument qui a été transmis lors de l'appel de la fonction.

À titre d'exemple un peu plus élaboré, nous allons maintenant modifier quelque peu la fonction `table()` sur laquelle nous avons déjà pas mal travaillé, afin qu'elle renvoie elle aussi une valeur. Cette valeur sera en l'occurrence une liste (la liste des dix premiers termes de la table de multiplication choisie). Voilà donc une occasion de reparler des listes. Dans la foulée, nous en profiterons pour apprendre encore un nouveau concept :

```
>>> def table(base):
...     resultat = [] # resultat est d'abord une liste vide
...     n = 1
...     while n < 11:
...         b = n * base
...         resultat.append(b) # ajout d'un terme à la liste
...         n = n +1 # (voir explications ci-dessous)
...     return resultat
...
```

Pour tester cette fonction, nous pouvons entrer par exemple :

```
>>> ta9 = table(9)
```

Ainsi nous affectons à la variable `ta9` les dix premiers termes de la table de multiplication par 9, sous la forme d'une liste :

```
>>> print(ta9)
[9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
>>> print(ta9[0])
9 >>> print(ta9[3])
36
>>> print(ta9[2:5])
[27, 36, 45]
>>>
```

Notes

— Comme nous l'avons vu dans l'exemple précédent, l'instruction return définit ce que doit être la valeur « renvoyée » par la fonction. En l'occurrence, il s'agit ici du contenu de la variable `resultat`, c'est-à-dire la liste des nombres générés par la fonction.

— L'instruction `resultat.append(b)` est notre second exemple de l'utilisation d'un concept important sur lequel nous reviendrons encore abondamment par la suite : dans cette instruction, nous appliquons la *méthode* `append()` à l'*objet* `resultat`. Nous préciserons petit à petit ce qu'il faut entendre par *objet* en programmation. Pour l'instant, admettons simplement que ce terme très général s'applique notamment aux listes de Python. Une *méthode* n'est en fait rien d'autre qu'une fonction (que vous pouvez d'ailleurs reconnaître comme telle à la présence des parenthèses), mais une fonction qui est associée à un objet. Elle fait partie de la définition de cet objet, ou plus précisément de la classe particulière à laquelle cet objet appartient (nous étudierons ce concept de classe plus tard).
Mettre en oeuvre une méthode associée à un objet consiste en quelque sorte à « faire fonctionner » cet objet d'une manière particulière. Par exemple, on met en oeuvre la méthode `methode4()` d'un objet `objet3`, à l'aide d'une instruction du type : `objet3.methode4()` , c'est-à-dire le nom de l'objet, puis le nom de la méthode, reliés l'un à l'autre par un point. Ce point joue un rôle essentiel : on peut le considérer comme un véritable opérateur.
Dans notre exemple, nous appliquons donc la méthode `append()` à l'objet `resultat`, qui est une liste. Sous Python, les listes constituent donc une classe particulière d'objets, auxquels on peut effectivement appliquer toute une série de méthodes. En l'occurrence, la méthode `append()` des objets « listes » sert à leur ajouter un élément par la fin. L'élément à ajouter est transmis entre les parenthèses, comme tout argument qui se respecte.

5.7 Utilisation des fonctions dans un script

Exemple

```
def cube(n):
    return n**3
```

```
def volumeSphere(r):
    return 4 * 3.1416 * cube(r) / 3

r = input('Entrez la valeur du rayon : ')
print('Le volume de cette sphère vaut', volumeSphere(float(r)))
```

À bien y regarder, ce programme comporte trois parties : les deux fonctions `cube()` et `volumeSphere()`, et ensuite le corps principal du programme.

Dans le corps principal du programme, on appelle la fonction `volumeSphere()`, en lui transmettant la valeur entrée par l'utilisateur pour le rayon, préalablement convertie en un nombre réel à l'aide de la fonction intégrée `float()`.

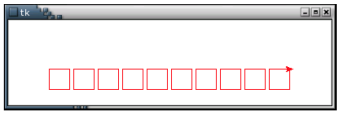
À l'intérieur de la fonction `volumeSphere()`, il y a un appel de la fonction `cube()`.

Notez bien que les trois parties du programme ont été disposées dans un certain ordre : d'abord la définition des fonctions, et ensuite le corps principal du programme. Cette disposition est nécessaire, parce que l'interpréteur exécute les lignes d'instructions du programme l'une après l'autre, dans l'ordre où elles apparaissent dans le code source.

5.8 Modules de fonctions

Afin que vous puissiez mieux comprendre encore la distinction entre la définition d'une fonction et son utilisation au sein d'un programme, nous vous suggérons de placer fréquemment vos définitions de fonctions dans un module Python, et le programme qui les utilise dans un autre.

Exemple : On souhaite réaliser la série de dessins ci-dessous, à l'aide du module `turtle` :



Ecrivez les lignes de code suivantes, et sauvegardez les dans un fichier auquel on donne le nom `dessins_tortue.py` :

```
from turtle import *

def carre(taille, couleur):
    "fonction qui dessine un carré de taille et de couleur déterminées"
    color(couleur)
    c =0
    while c <4:
        forward(taille)
        right(90)
    c = c +1
```

ne joue aucun rôle fonctionnel dans le script : elle est traitée par Python comme un simple *commentaire*, mais qui est mémorisé à part dans un système de documentation interne automatique, lequel pourra ensuite être exploité par certains utilitaires et éditeurs « intelligents ».

En fait, Python place cette chaîne dans une variable spéciale dont le nom est `__doc__` (le mot « doc » entouré de deux paires de caractères « souligné »), et qui est associée à l'objet fonction comme étant l'un de ses attributs.

```
>>> print(carre.__doc__)
fonction qui dessine un carré de taille et de couleur déterminées
```

Le fichier que vous aurez créé ainsi est dorénavant un véritable module de fonctions Python, au même titre que les modules turtle ou math que vous connaissez déjà. Vous pouvez donc l'utiliser dans n'importe quel autre script, comme celui-ci, par exemple, qui effectuera le travail demandé :

```
from dessins_tortue import *

up() # relever le crayon
goto(-150, 50) # reculer en haut à gauche

# dessiner dix carrés rouges, alignés :
i = 0
while i < 10:
    down() # abaisser le crayon
    carre(25, 'red') # tracer un carré
    up() # relever le crayon
    forward(30) # avancer + loin
    i = i +1

a = input() # attendre
```

5.9 Valeurs par défaut pour les paramètres

Dans la définition d'une fonction, il est possible (et souvent souhaitable) de définir un argument par défaut pour chacun des paramètres. On obtient ainsi une fonction qui peut être appelée avec une partie seulement des arguments attendus. Exemples :

```
>>> def politesse(nom, vedette ='Monsieur'):
...     print("Veuillez agréer ,", vedette, nom, ", mes salutations cordiales.")
...
>>> politesse('Dupont')
Veuillez agréer , Monsieur Dupont , mes salutations cordiales.
>>> politesse('Durand', 'Mademoiselle')
Veuillez agréer , Mademoiselle Durand , mes salutations cordiales.
```

Lorsque l'on appelle cette fonction en ne lui fournissant que le premier argument, le second reçoit tout de même une valeur par défaut. Si l'on fournit les deux arguments, la valeur par défaut

pour le deuxième est tout simplement ignorée. Vous pouvez définir une valeur par défaut pour tous les paramètres, ou une partie d'entre eux seulement. Dans ce cas, cependant, les paramètres sans valeur par défaut doivent précéder les autres dans la liste.

Exercices

1. *Tri par sélection*

   (a) *Ecrire une fonction* `indMin` *on renvoie l'indice du minimum (en renvoyant le plus petit s'il en existe plusieurs). Par exemple,* `indMin([5,2,3,4,2])` *renvoie 1 (les éléments sont indicés à partir de 0).*

   (b) *Écrire une procédure* `echange` *qui étant donnés une liste* `a` *et deux entiers* `i` *et* `j` *permute dans a les éléments d'indices* `i` *et* `j` *. Par exemple :*

   ```
   >>>a = [1,2,3,4,5]
   >>>echange(a,0,2)
   >>>print(a)
   [3,2,1,4,5]
   ```

   (c) *Ecrire une procédure* `tri` *qui étant donnée une liste* `a` *modifie cette liste de sorte à trier ses éléments (sans modifier l'ensemble de ses valeurs). On utilisera un tri par sélection (des minima) : Le principe de l'algorithme consiste à déterminer la position du plus petit élément et à le mettre en première position (par un échange), puis d'itérer le procédé sur le sous-tableau restant. Il faut* $(n-1)$ *comparaisons pour déterminer la position du plus petit élément, donc il faut* $\frac{n}{2}(n-1)$ *comparaisons pour trier une liste selon ce procédé.*

2. *Tests de permutation*
   Il s'agit de déterminer si une liste de longueur n correspond à une permutation de $\{0, \cdots, n-1\}$ *, c'est-à-dire si tout entier compris entre 0 et* $n-1$
   apparaît une et une seule fois dans la liste. Il s'agit d'écrire une fonction à valeurs booléennes qui étant donnée une liste `a` *de longueur n renvoie* `True` *ssi* `a` *est une permutation de*$\{0, \cdots, n-1\}$.
   On propose deux méthodes, la seconde étant plus efficace que la première (le nombre d'opérations est moindre).

   (a) *Ecrire une première fonction* `test1` *qui pour tout entier* $i \in \{0, \cdots, n-1\}$*, vérifie qu'il existe un unique élément de la liste* `a` *valant* j*, et interrompt la procédure dès que ce test est faux. La complexité dans le pire des cas est en* $O(n^2)$.

   (b) *Ecrire une première fonction* `test2` *qui pour tout entier* $i \in \{0, \cdots, n-1\}$*, construit en temps linéaire* $O(n)$ *le tableau* `b` *des occurrences, c'est-à-dire que* $b[j]$ *vaut le nombre de termes de* `a` *valant* j*. On vérifie ensuite (en temps linéaire* $O(n)$*) que le tableau des occurrences ne contient que des 1. La complexité est* $O(n)$ *.*
   Remarque : une instruction comportant un `return` *permet d'interromre une boucle* `for` *.*

A l'avenir, on pourra utiliser la fonction `search` du module `re`. Pas exemple,

```
>>> import re
>>> animaux = "girafe tigre singe"
>>> re.search("tigre", animaux)
<_sre.SRE_Match object at 0x7fefdaefe2a0>
>>> if re.search("tigre", animaux):
...     print "OK"
...
OK
```

A retenir

- L'approche efficace d'un problème complexe consiste souvent à le décomposer en plusieurs sous-problèmes plus simples qui seront étudiés séparément. Il est important que cette décomposition soit représentée fidèlement dans les algorithmes pour que ceux-ci restent clairs.
- Les *fonctions* et les *classes d'objets* permettent de structurer les programmes en Python.
- Structure d'une fonction

```
def nomDeLaFonction(liste de paramètres):
    ...
    bloc d'instructions
    ...
    return liste
```

- On peut affecter des valeurs par défaut aux paramètres d'une fonction. A l'appel de la fonction, on peut alors omettre ces paramètres.
- A l'intérieur du corps d'une fonction les variables sont des *variables locales*.
- Python est composé de nombreux modules proposant des fonctions prédéfinies.

Chapitre 6

Classes, objets, attributs

6.1 Utilité des classes

Les classes sont les principaux outils de la programmation orientée objet (Object Oriented Programming ou OOP). Ce type de programmation permet de structurer les logiciels complexes en les organisant comme des ensembles d'objets qui interagissent, entre eux et avec le monde extérieur.

Le premier bénéfice de cette approche de la programmation réside dans le fait que les différents objets utilisés peuvent être construits indépendamment les uns des autres (par exemple par des programmeurs différents) sans qu'il n'y ait de risque d'interférence. Ce résultat est obtenu grâce au concept d'*encapsulation* : la fonctionnalité interne de l'objet et les variables qu'il utilise pour effectuer son travail, sont en quelque sorte « enfermées » dans l'objet. Les autres objets et le monde extérieur ne peuvent y avoir accès qu'à travers des procédures bien définies : *l'interface de l'objet*.

En particulier, l'utilisation de classes dans vos programmes va vous permettre - entre autres avantages -d'éviter au maximum l'emploi de *variables globales*. Vous devez savoir en effet que l'utilisation de variables globales comporte des risques, d'autant plus importants que les programmes sont volumineux, parce qu'il est toujours possible que de telles variables soient modifiées, ou même redéfinies, n'importe où dans le corps du programme (ce risque s'aggrave particulièrement si plusieurs programmeurs différents travaillent sur un même logiciel).

Un second bénéfice résultant de l'utilisation des classes est la possibilité qu'elles offrent de construire de nouveaux objets à partir d'objets préexistants, et donc de réutiliser des pans entiers d'une programmation déjà écrite (sans toucher à celle-ci !), pour en tirer une fonctionnalité nouvelle. Cela est rendu possible grâce aux concepts de *dérivation* et de *polymorphisme* :

— La *dérivation* est le mécanisme qui permet de construire une classe « enfant » au départ d'une classe « parente ». L'enfant ainsi obtenu hérite toutes les propriétés et toute la fonctionnalité de son ancêtre, auxquelles on peut ajouter ce que l'on veut.

— Le *polymorphisme* permet d'attribuer des comportements différents à des objets dérivant les uns des autres, ou au même objet ou en fonction d'un certain contexte.

Avant d'aller plus loin, signalons ici que la programmation orientée objet est *optionnelle* sous Python. Vous pouvez donc mener à bien de nombreux projets sans l'utiliser, avec des outils plus simples tels que les fonctions. Sachez cependant que si vous faites l'effort d'apprendre à programmer à l'aide de classes, vous maîtriserez un niveau d'abstraction plus élevé, ce qui vous permettra de traiter des problèmes de plus en plus complexes.

6.2 Définition d'une classe élémentaire

Pour créer une nouvelle classe d'objets Python, on utilise l'instruction `class`. Nous allons donc apprendre à utiliser cette instruction, en commençant par définir un type d'objet très rudimentaire, lequel sera simplement un nouveau type de donnée. Nous avons déjà utilisé différents types de données jusqu'à présent, mais il s'agissait à chaque fois de types intégrés dans le langage lui-même. Nous allons maintenant créer un nouveau type composite : le type `Point`. Ce type correspondra au concept de point en géométrie plane. Dans un plan, un point est caractérisé par deux nombres (ses coordonnées suivant x et y). En notation mathématique, on représente donc un point par ses deux coordonnées x et y enfermées dans une paire de parenthèses. On parlera par exemple du point (25, 17). Une manière naturelle de représenter un point sous Python serait d'utiliser pour les coordonnées deux valeurs de type float. Nous voudrions cependant combiner ces deux valeurs dans une seule entité, ou un seul objet. Pour y arriver, nous allons définir une classe `Point()` :

```
>>> class Point(object):
...     "Définition d'un point géométrique"
```

Les définitions de classes peuvent être situées n'importe où dans un programme, mais on les placera en général au début (ou bien dans un module à importer). L'exemple ci-dessus est probablement le plus simple qui se puisse concevoir. Une seule ligne nous a suffi pour définir le nouveau type d'objet `Point()`.

Remarquons d'emblée que :

— L'instruction class est un nouvel exemple d'instruction composée. N'oubliez pas le double point obligatoire à la fin de la ligne, et l'indentation du bloc d'instructions qui suit. Ce bloc doit contenir au moins une ligne. Dans notre exemple ultra-simplifié, cette ligne n'est rien d'autre qu'un simple commentaire. Comme nous l'avons vu précédemment pour les fonctions, vous pouvez insérer une chaîne de caractères directement après l'instruction `class`, afin de mettre en place un commentaire qui sera automatiquement incorporé dans le dispositif de documentation interne de Python. Prenez donc l'habitude de toujours placer une chaîne décrivant la classe à cet endroit.

— Les parenthèses sont destinées à contenir la référence d'une classe préexistante. Cela est requis pour permettre le mécanisme d'héritage. Toute classe nouvelle que nous créons peut en effet hériter d'une classe parente un ensemble de caractéristiques, auxquelles elle ajoutera les siennes propres. Lorsque l'on désire créer une classe fondamentale - c'est-à-dire ne dérivant d'aucune autre, comme c'est le cas ici avec notre classe `Point()` - la référence à indiquer doit être par convention le nom spécial object, lequel désigne l'ancêtre de toutes les classes.

— Une convention très répandue veut que l'on donne aux classes des noms qui commencent par une majuscule. Dans la suite de ce texte, nous respecterons cette convention, ainsi qu'une autre qui demande que dans les textes explicatifs, on associe à chaque nom de classe une paire de parenthèses, comme nous le faisons déjà pour les noms de fonctions.

Nous venons donc de définir une classe `Point()`. Nous pouvons à présent nous en servir pour créer des objets de cette classe, que l'on appellera aussi des instances de cette classe. L'opération s'appelle pour cette raison une *instanciation*. Créons par exemple un nouvel objet `p9` :

```
>>> p9 = Point()
```

Après cette instruction, la variable p9 contient la référence d'un nouvel objet Point(). Nous pouvons dire également que p9 est une nouvelle *instance* de la classe verb=Point()=.

Voyons maintenant si nous pouvons faire quelque chose avec notre nouvel objet p9 :

```
>>> print(p9)
<__main__.Point object at 0xb76f132c>
```

Le message renvoyé par Python indique, comme vous l'aurez certainement bien compris tout de suite, que p9 est une instance de la classe Point(), laquelle est définie elle-même au niveau principal (main) du programme. Elle est située dans un emplacement bien déterminé de la mémoire vive, dont l'adresse apparaît ici en notation hexadécimale.

```
>>> print(p9.__doc__)
Définition d'un point géométrique
```

6.3 Attributs (ou variables) d'instance

L'objet que nous venons de créer est juste une coquille vide. Nous allons à présent lui ajouter des composants, par simple assignation, en utilisant le système de qualification des noms par points :

```
>>> p9.x = 3.0
>>> p9.y = 4.0
```

Les variables x et y que nous avons ainsi définies en les liant d'emblée à p9 , sont désormais des attributs de l'objet p9. On peut également les appeler des variables d'instance. Elles sont en effet incorporées, ou plutôt encapsulées dans cette instance (ou objet). Le diagramme d'état ci-contre montre le résultat de ces affectations : la variable p9 contient la référence indiquant l'emplacement mémoire du nouvel objet, qui contient lui-même les deux attributs x et y. Ceux-ci contiennent les références des valeurs 3.0 et 4.0, mémorisées ailleurs.

```
>>> print(p9.x)
3.0
>>> print(p9.x**2 + p9.y**2)
25.0
```

Une approche classique pour créer des attributs consiste à utiliser les fonctions. On parle aussi de *constructeur*. Nous allons créer une classe Personne et ses attributs nom, prenom, etc.. Nous allons utiliser une méthode spéciale, appelée un constructeur, qui est appelée invariablement quand on souhaite créer un objet depuis notre classe.

Concrètement, un constructeur est une méthode de notre objet se chargeant de créer nos attributs. En vérité, c'est même la méthode qui sera appelée quand on voudra créer notre objet.

Voyons le code, ce sera plus parlant :

```
class Personne:
    """Classe définissant une personne caractérisée par :
    - son nom
    - son prénom
    - son âge
    - son lieu de résidence"""


    def __init__(self): # Notre méthode constructeur
        """Constructeur de notre classe. Chaque attribut va être instancié
        avec une valeur par défaut... original"""


        self.nom = "Dupont"
        self.prenom = "Jean" # Quelle originalité
        self.age = 33 # Cela n'engage à rien
        self.lieu_residence = "Paris"
```

— La définition de notre constructeur. Comme vous le voyez, il s'agit d'une définition presque « classique » d'une fonction. Elle a pour nom `__init__`, c'est invariable : en Python, tous les constructeurs s'appellent ainsi. Nous verrons plus tard que les noms de méthodes entourés de part et d'autre de deux signes soulignés (`__nommethode__`) sont des méthodes spéciales. Notez que, dans notre définition de méthode, nous passons un premier paramètre nommé self.

— Dans notre constructeur, nous trouvons l'instanciation de notre attribut `nom`. On crée une variable self.nom et on lui donne comme valeur `Dupont`.

```
>>> bernard = Personne() # permet de créer l'objet
>>> bernard # renvoie l'adresse de l'objet
<__main__.Personne object at 0x00B42570>
>>> bernard.nom
'Dupont'
>>>
```

Bon. Il nous reste encore à faire un constructeur un peu plus intelligent. Pour l'instant, quel que soit l'objet créé, il possède les mêmes nom, prénom, âge et lieu de résidence. On peut les modifier par la suite, bien entendu, mais on peut aussi faire en sorte que le constructeur prenne plusieurs paramètres, disons… le nom et le prénom, pour commencer.

```
class Personne:
    """Classe définissant une personne caractérisée par :
    - son nom
    - son prénom
    - son âge
```

```
    - son lieu de résidence"""


    def __init__(self, nom, prenom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.prenom = prenom
        self.age = 33
        self.lieu_residence = "Paris"
```

Nous pouvons alors utiliser la class :

```
>>> bernard = Personne("Micado", "Bernard")
>>> bernard.nom
'Micado'
>>> bernard.prenom
'Bernard'
>>> bernard.age
33
>>>
```

Vous avez remarqué qu'on utilise un objet `self`. En pratique, cet objet `self` est l'objet qui appelle la méthode.

Exercice
comprendre et expliquer ce que fait le programme suivant

```
# CompteBancaire.py

# définition de la classe Compte
class Compte:
   """Un exemple de classe :
   gestion d'un compte bancaire"""

   # définition de la méthode spéciale __init__ (constructeur)
   def __init__(self,soldeInitial):
      """Initialisation du compte avec la valeur soldeInitial."""
      # assignation de l'attribut d'instance solde
      self.solde = float(soldeInitial)

   # définition de la méthode NouveauSolde()
   def NouveauSolde(self,somme):
      """Nouveau solde de compte avec la valeur somme."""
      self.solde = float(somme)

   # définition de la méthode Solde()
   def Solde(self):
      """Retourne le solde."""
      return self.solde

   # définition de la méthode Credit()
   def Credit(self,somme):
      """Crédite le compte de la valeur somme. Retourne le solde."""
      self.solde += somme
      return self.solde

   # définition de la méthode Debit()
   def Debit(self,somme):
      """Débite le compte de la valeur somme. Retourne le solde."""
      self.solde -= somme
      return self.solde

   # définition de la méthode spéciale __add__ (surcharge de l'opérateur +)
   def __add__(self,somme):
      """x.__add__(somme) <=> x+somme
      Crédite le compte de la valeur somme.
      Affiche 'Nouveau solde : somme'"""
      self.solde += somme
      print("Nouveau solde : {:+.2f} euros".format(self.solde))
   # définition de la méthode spéciale __sub__ (surcharge de l'opérateur -)
   def __sub__(self,somme):
      """x.__sub_(somme) <=> x-somme
      Débite le compte de la valeur somme.
      Affiche 'Nouveau solde : somme'"""
      self.solde -= somme
      print("Nouveau solde : {:+.2f} euros".format(self.solde))

if __name__ == '__main__':
```

6.4 Passage d'objets comme arguments dans l'appel d'une fonction

Les fonctions peuvent utiliser des objets comme paramètres, et elles peuvent également fournir un objet comme valeur de retour. Par exemple, vous pouvez définir une fonction telle que celle-ci :

```
>>> def affiche_point(p):
... print("coord. horizontale =", p.x, "coord. verticale =", p.y)
```

Le paramètre `p` utilisé par cette fonction doit être un objet de type `Point()`, dont l'instruction qui suit utilisera les variables d'instance `p.x` et `p.y`. Lorsqu'on appelle cette fonction, il faut donc lui fournir un objet de type `Point()` comme argument. Essayons avec l'objet `p9` :

```
>>> affiche_point(p9)
coord. horizontale = 3.0 coord. verticale = 4.0
```

Exercice
Écrivez une fonction `distance()` *qui permette de calculer la distance entre deux points. (Il faudra vous rappeler le théorème de Pythagore !) Cette fonction attendra évidemment deux objets* `Point()` *comme arguments.*

6.5 Similitude et unicité

Dans la langue parlée, les mêmes mots peuvent avoir des significations fort différentes suivant le contexte dans lequel on les utilise. La conséquence en est que certaines expressions utilisant ces mots peuvent être comprises de plusieurs manières différentes (expressions ambiguës).

Le mot « même », par exemple, a des significations différentes dans les phrases : « Charles et moi avons la même voiture » et « Charles et moi avons la même mère ». Dans la première, ce que je veux dire est que la voiture de Charles et la mienne sont du même modèle. Il s'agit pourtant de deux voitures distinctes. Dans la seconde, j'indique que la mère de Charles et la mienne constituent en fait une seule et unique personne.

Lorsque nous traitons d'objets logiciels, nous pouvons rencontrer la même ambiguïté. Par exemple, si nous parlons de l'égalité de deux objets `Point()`, cela signifie-t-il que ces deux objets contiennent les mêmes données (leurs attributs), ou bien cela signifie-t-il que nous parlons de deux références à un même et unique objet ?

Considérez par exemple les instructions suivantes :

```
>>> p1 = Point()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Point()
>>> p2.x = 3
>>> p2.y = 4
>>> print(p1 == p2)
False
```

Ces instructions créent deux objets `p1` et `p2` qui restent distincts, même s'ils font partie d'une même classe et ont des contenus similaires. La dernière instruction teste l'égalité de ces deux objets (double signe égale), et le résultat est `False` : il n'y a donc pas égalité.

Essayons autre chose, à présent :

```
>>> p2 = p1
>>> print(p1 == p2)
True
```

Par l'instruction `p2  p1=` , nous assignons le contenu de `p1` à `p2`. Cela signifie que désormais ces deux variables référencent le même objet. Les variables `p1` et `p2` sont des $alias$ l'une de l'autre.

A retenir

- Les classes, comme les fonctions, permettent de structurer efficacement les programmes.
- Les classes définissent des objets auxquels on pu attacher des attributs.

Chapitre 7

Modules existants : matplotlib, numpy

7.1 numpy

Le module `numpy` est incontournable en mathématique. Il permet d'effectuer des calculs sur des vecteurs ou des matrices, élément par élément, via un nouveau type d'objet appelé array. Ce module contient des fonctions de base pour faire de l'algèbre linéaire, des transformées de Fourier ou encore des tirages de nombres aléatoires plus sophistiqués qu'avec le module `random`. Vous pourrez trouver les sources de `numpy` à cette adresse. Notez qu'il existe un autre module `scipy` que nous n'aborderons pas dans ce cours. `scipy` est lui-même basé sur `numpy`, mais il en étend considérablement les possibilités de ce dernier (e.g. statistiques, optimisation, intégration numérique, traitement du signal, traitement d'image, algorithmes génétiques, etc.).

7.1.1 Objets de type array

Les objets de type `array` correspondent à des tableaux à une ou plusieurs dimensions et permettent d'effectuer du calcul vectoriel. La fonction `array()` permet la conversion d'un objet séquentiel (type liste ou tuple) en un objet de type array. Voici un exemple simple de conversion d'une liste à une dimension en objet `array` :

```
>>> import numpy as np
>>> a = [1,2,3]
>>> numpy.array(a)
array([1, 2, 3])
>>> b = np.array(a)
>>> type(a)
<type np.ndarray>
>>> b
array([1, 2, 3])
```

La différence fondamentale entre un objet array à une dimension et une liste (ou un tuple) est que celui-ci est considéré comme un vecteur. Par conséquent on peut effectuer des opérations élément par élément dessus, ce qui est bien commode lorsqu'on analyse de grandes quantités de données. Regardez ces exemples :

```
>>> v = np.arange(4)
>>> v
array([0, 1, 2, 3])
>>> v + 1
array([1, 2, 3, 4])
>>> v + 0.1
array([ 0.1, 1.1, 2.1, 3.1])
>>> v * 2
array([0, 2, 4, 6])
>>> v * v
array([0, 1, 4, 9])
```

Notez bien sur le dernier exemple de multiplication que l'array final correspond à la multiplication élément par élément des deux array initiaux. Avec les listes, ces opérations n'auraient été possibles qu'en utilisant des boucles ! Nous vous encourageons donc à utiliser dorénavant les objets array lorsque vous aurez besoin de faire des opérations élément par élément.

Il est aussi possible de construire des objets array à deux dimensions, il suffit de passer en argument une liste de listes à la fonction `array()` :

```
>>> np.array([[1,2,3],[2,3,4],[3,4,5]])
array([[1, 2, 3],
[2, 3, 4],
[3, 4, 5]])
>>>
```

Dans le cas d'un objet array à deux dimensions, vous pouvez récupérer une ligne, une colonne ou bien un seul élément.

```
>>> a = np.array([[1,2],[3,4]])
>>> a
array([[1, 2],
   [3, 4]])
>>> a[:,0]
array([1, 3])
>>> a[0,:]
array([1, 2])
>>> a[1,1]
4
```

Un tableau numpy appartient à une classe et des attributs lui son attachés. Par exemple

```
>>> a = np.array([[1,2,7],[3,4,5]])
>>> a.shape
(2, 3)
```

```
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize

>>> a.size
```

La fonction `dot()` vous permet de faire une multiplication de matrices.

```
>>> a = np.resize(np.arange(4),(2,2))
>>> a
array([[0, 1],
[2, 3]])
>>> np.dot(a,a)
array([[ 2, 3],
[ 6, 11]])
>>> a * a
array([[0, 1],
[4, 9]])
```

Pour toutes les opérations suivantes, il faudra utiliser des fonctions dans le sous-module `np.linalg`. La fonction `inv()` renvoie l'inverse d'une matrice carrée, `det()` son déterminant, `eig()` ses vecteurs et valeurs propres.

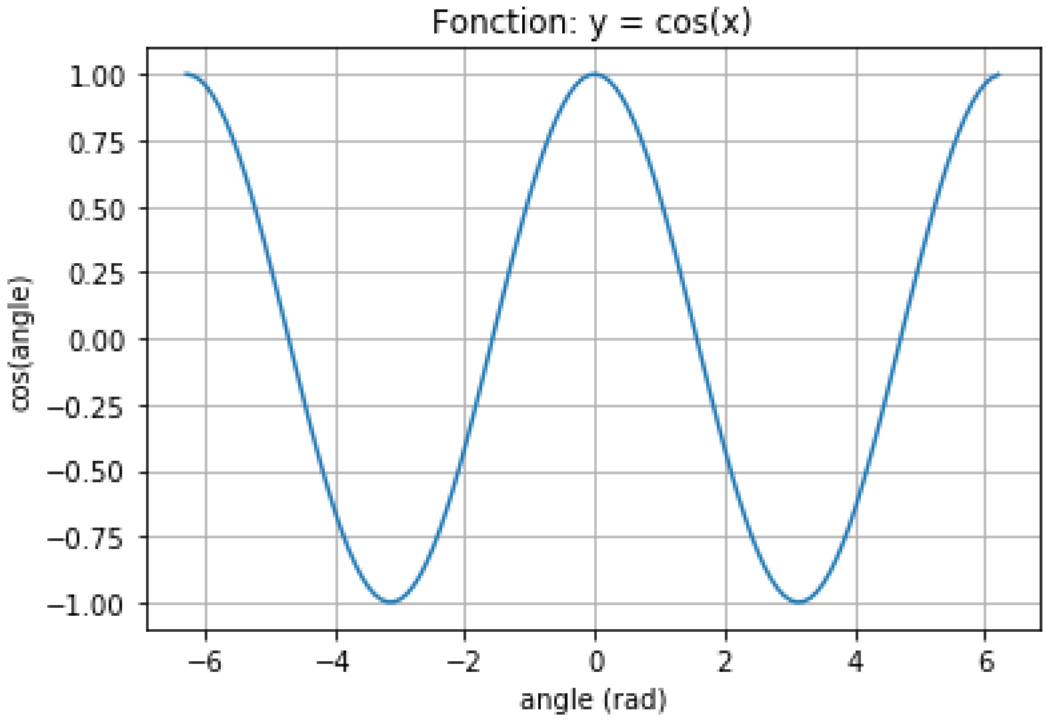
```
>>> a
array([[0, 1],
    [2, 3]])
>>> np.linalg.inv(a)
array([[-1.5, 0.5],
    [ 1. , 0. ]])
>>> np.linalg.det(a)
-2.0
>>> np.linalg.eig(a)
(array([-0.56155281, 3.56155281]), array([[-0.87192821, -0.27032301],
    [ 0.48963374, -0.96276969]]))
>>> np.linalg.eig(a)[0]
array([-0.56155281, 3.56155281])
>>> np.linalg.eig(a)[1]
array([[-0.87192821, -0.27032301],
    [ 0.48963374, -0.96276969]])
```

7.2 Module matplotlib

Le module `matplotlib` permet de générer des graphes interactifs depuis Python. Il est l'outil complémentaire de `numpy` et `scipy` lorsqu'on veut faire de l'analyse de données.

```
# define cosine function
from pylab import *
debut = -2 * pi
fin = 2 * pi
pas = 0.1
x = arange(debut,fin,pas)
y = cos(x)
# draw the plot
plot(x,y)
xlabel("angle (rad)")
ylabel("cos(angle)")
title("Fonction: y = cos(x)")
grid()
show()
```

Vous devriez obtenir une image comme celle-ci :



— La fonction `plot()` va générer un graphique avec des lignes et prend comme valeurs en abscisse (x) et en ordonnées (y) des vecteurs de type array à une dimension.
— Les fonctions `xlabel()` et `ylabel()` sont utiles pour donner un nom aux axes.
— `title()` permet de définir le titre du graphique.

— `grid()` affiche une grille en filigrane.
— Jusqu'ici, aucun graphe n'est affiché. Pour activer l'affichage à l'écran du graphique, il faut appeler la fonction `show()`. Celle-ci va activer une boucle dite gtkqui attendra les manipulations de l'utilisateur.