1 Qu'est-ce qu'un langage de programmation ?

Programmer consiste à demander à l'ordinateur d'effectuer des actions, comme par exemple *Faire le calcul 3 + 5*. Comme on l'a déjà vu, un ordinateur ne peut stocker et travailler qu'en langage binaire (informations codées par des suites de 0 et de 1). Il est inconcevable d'écrire directement un programme en langage binaire, car ce serait extrêmement fastidieux !

Pour résoudre ce problème, on écrit les programmes dans des langages évolués, *i.e.*, compréhensibles par le programmeur comme, par exemple, le langage `C` ou le langage `Python` qui sont des *langages évolués de haut niveau*. En informatique, un langage de programmation est une notation conventionnelle destinée à formuler des algorithmes et produire des programmes informatiques qui les appliquent. D'une manière similaire à une langue naturelle, un langage de programmation est composé d'un alphabet, d'un vocabulaire et de règles de grammaire.

Comme l'ordinateur ne travaille qu'en *langage binaire ou machine*, il ne comprend pas et ne peut exécuter directement un langage évolué. Il faut donc traduire ce programme écrit en langage évolué en langage binaire ou en langage machine : c'est le rôle de la *"compilation"* ou de *"l'interprétation"*. Les programmes réalisant cette traduction automatique s'appellent des *"compilateurs"* ou des *"interpréteurs"*. Le programme produit par un compilateur s'appelle un *exécutable* : il pourra ensuite être directement exécuté par un ordinateur. Au contraire, l'interpréteur réalise cette traduction « à la volée » (il ne produit pas d'exécutable).

Pour un programme écrit, par exemple, en langage `C`, on utilise un *compilateur* `C`. On dit alors que le langage `C` est un *langage compilé*. En revanche, pour un programme écrit en `Python`, on utilise un *interpréteur* `Python` qui traduit le le code source ligne par ligne en langage machine au moment même de l'exécution du programme. On dit dans ce cas que la langage `Python` est un *langage interprété*. En d'autres termes, un *compilateur* traduit l'ensemble du programme avant qu'il puisse être exécuté et fournit un programme binaire à la suite de ce travail. En revanche, un *interpréteur* traduit une ligne et le code machine correspondant est exécuté ; puis il traduit ensuite la ligne suivante qui peut alors être exécutée, etc. En particulier, le code binaire associé n'est jamais disponible car jamais enregistré.

2 Un peu d'histoire

Entre 1842 et 1843, une jeune comtesse du nom d'Ada Lovelace traduisait le mémoire d'un mathématicien italien du nom de Luigi Menabrea sur la machine analytique proposée par Charles Babbage. À cette traduction, la jeune comtesse avait ajouté ses propres notes dont l'une décrivait de façon détaillée une séquence progressive d'opérations pour résoudre certains problèmes mathématiques. Le premier programme était né.

Dans les années 1950, les trois premiers langages de programmation modernes ont été conçus :

- `FORTRAN`, un traducteur de formules (FORmula TRANslator),
- `LISP`, spécialisé dans le traitement des listes (LISt Processor),
- `COBOL`, spécialisé dans la programmation d'application de gestion (COmmon Business Oriented Langage).

En 1972 le C fait son apparition. Créé par le regretté Denis Ritchie, ce langage a servi à coder le système Unix.

Le langage `Python` est un langage *interprété* qui permet (sans l'imposer) une approche modulaire et orientée objet de la programmation. Il est développé depuis 1991 par Guido van Rossum (université d'Amsterdam) et de nombreux contributeurs bénévoles. Il a été nommé ainsi en référence à la série télévisée *Monthy Python's Flying Circus*.

Les principales caractéristiques de `Python` sont les suivantes :

- Il est *open source*. Libre et gratuit, il est supporté, développé et utilisé par une large communauté : 300 000 utilisateurs et plus de 500 000 téléchargements par an. Il peut également être utiliser sans restriction dans des projets commerciaux.
- Il est *portable* non seulement sur les différentes variantes `Unix`, mais aussi sur les `OS` propriétaires : `Mac OS`, `Windows`, etc...
- Il possède *une syntaxe 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].
- Il gère ses ressources (mémoire, descripteurs de fichiers, etc...) sans intervention du programmeur par un mécanisme de *comptage de référence*.
- Il n'y a pas de *pointeurs explicites* comme dans la plupart des langages de programmation.
- Il est (optionnellement) *multi-threadé*, c'est-à-dire qu'il est capable d'exécuter efficacement plusieurs tâches simultanément.
- Il est *orienté objet* : Il supporte *l'héritage multiple* et *la surcharge des opérateurs*. Dans son modèle objet, et en reprenant la terminologie de `C++`, toutes les méthodes sont virtuelles.
- Il intègre, comme `Java` ou les versions récentes de `C++`, un système *d'exceptions*, qui permettent de simplifier considérablement la gestion des erreurs.
- Il est *dynamique* (l'interpréteur peut évaluer des chaînes de caractères représentant des expressions ou des instructions), *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).
- Il est *dynamiquement typé* : tout objet manipulable par le programmeur possède un type bien défini à l'exécution, qui n'a pas besoin d'être déclaré à l'avance.
- Il est *extensible* : on peut facilement l'interfacer avec des bibliothèques `C` existantes et on peut aussi s'en servir comme d'un langage d'extension pour des systèmes logiciels complexes.
- Sa *bibliothèque standard* et les *paquetages contribués*, donnent accès à une grande variété de services : calcul scientifique, chaînes de caractères et expressions régulières, services `UNIX` standards (fichiers, pipes, signaux, sockets, threads, etc...), protocoles Internet (Web, News, `FTP`, `CGI`, `HTML`, etc...), persistance et bases de données, interfaces graphiques, etc...

Le langage `Python` continue à évoluer, soutenu par une grande communauté d'utilisateurs, 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 second interpréteur, écrit en `Java`, est en cours de développement.

Actuellement, les programmes peuvent être écrits en `Python2` ou `Python3`. Les versions les plus récentes sont `Python3.8` (stable) et `Python3.9` (en développement). On renvoie au site officiel de `Python` [2] pour plus de détails.

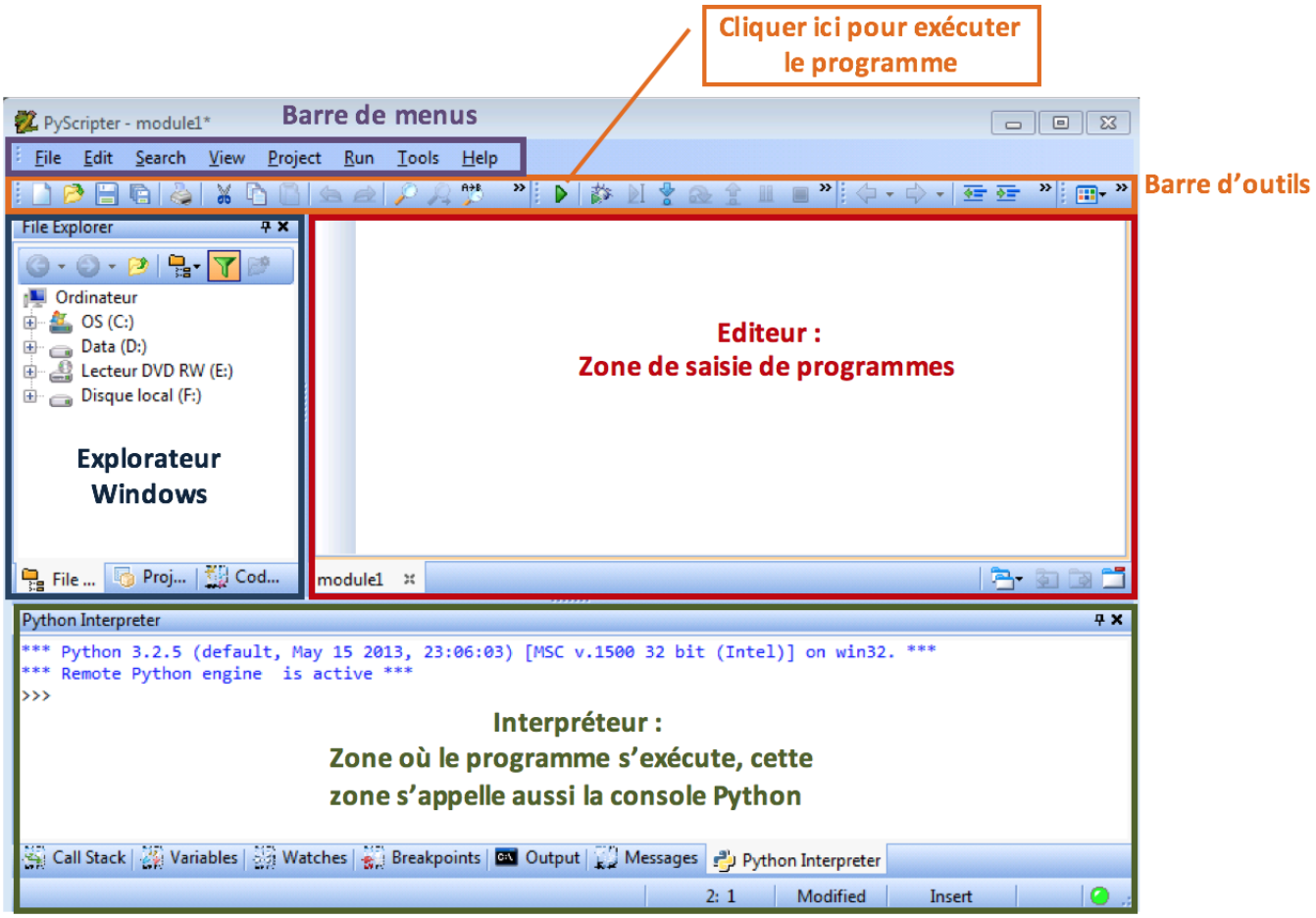
Toutes les instructions décrites dans ce document sont en `Python3`.

3 Prise en main de `Python`

Pour l'enseignement de spécialité NSI, nous avons choisi de programmer en langage `Python`, version 3.4, avec le logiciel `EduPython` fonctionnant sous `Windows` :

Après avoir lancé le logiciel, il se présente de la façon suivante :

1. A fonctionnalités égales, un programme `Python` (abondamment commenté et présenté selon les canons standards) est souvent de 3 à 5 fois plus court qu'un programme `C` ou `C++` (ou même `Java`) équivalent, ce qui représente en général un temps de développement de 5 à 10 fois plus court et une facilité de maintenance largement accrue.
2. https://www.python.org/doc/



La fenêtre graphique du logiciel `EduPython` est divisé en plusieurs zones :

- une barre de menu en haut,
- une barre d'outils placée juste en dessous de la barre de menu,
- une zone pour l'explorateur de fichiers de `Windows`,
- une zone d'édition de programmes,
- une console d'exécution.

4 Opérations numériques usuelles

4.1 Opérations algébriques

addition	soustraction	multiplication	division	puissance n^{ème} de x
`+`	`-`	`*`	`/`	`x**n`

Remarque 4.1 Attention, les opérations doivent toujours être écrites. Par exemple, on écrit `2*x` et non pas `2x`.

4.2 Opérations avec les nombres entiers

Soient `a` et `b` deux nombres entiers avec $b \neq 0$. Il existe un unique couple d'entiers naturels (q,r) tels que $a = b \times q + r$ avec $0 \leqslant r < b$. Les nombres q et r s'appellent respectivement *le quotient* et *le reste de la division euclidienne de a par b*.

quotient de `a` par `b`	reste de `a` par `b`
`a//b`	`a%b`

Exemple 4.2 L'instruction `5//2` renvoie `2`, le quotient de la division euclidienne de 5 par 2 et l'instruction `5%2` renvoie `1`, le reste de la division euclidienne de 5 par 2.

4.3 Valeur absolue et arrondis

valeur absolue de x	arrondi de x avec n décimales
`abs(x)`	`round(x,n)`

Exemple 4.3 L'instruction `vabs(-5,3)` renvoie 5,3, la distance à zéro du nombre $-5{,}3$.
L'instruction `round(5/3)` renvoie 2 (arrondi à l'entier) et l'instruction `round(5/3,3)` renvoie 1,667 (arrondi au millième).

4.4 Avec la bibliothèque `math`

Le langage `Python` est muni de *bibliothèques* ou *modules* que l'on peut importer au besoin. Par exemple, la bibliothèque `math` comporte un grand nombre de fonctions mathématiques, comme la racine carrée, le sinus, le cosinus, ... On peut charger la bibliothèque `math` avec l'une ou l'autre des instructions suivantes :

```
from math import *
```

```
import math
```

Dans le second cas on fait appel aux fonctions de la bibliothèque en les précédant du préfixe `math.`.

Exemple 4.4 Les instructions suivantes permettent de calculer la racine carrée de 2 :

```
>>> from math import *
>>> sqrt(2)
1.4142135623730951
```

```
>>> import math
>>> math.sqrt(2)
1.4142135623730951
```

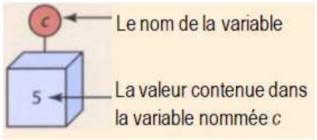
Remarque 4.5 on peut également charger uniquement une ou plusieurs fonctions choisies de la façon suivante :

```
from math import nomFonction1, nomFonction2
```

Par exemple, dans le cas de la fonction `sqrt` on saisirait alors `from math import sqrt` puis `sqrt(2)`.

5 Variables et affectations

Définition 5.1 (Variables) Une *variable* est désignée par un nom et fait référence à un emplacement dans la mémoire de l'ordinateur. Elle contient une *valeur* qui peut évoluer au cours de l'exécution de l'algorithme.



Autrement dit, une variable informatique peut-être vue comme une boîte avec un nom (une étiquette) qui peut contenir différentes valeurs. Au fur et à mesure de l'exécution d'un algorithme, le contenu de la variable est susceptible d'être modifié.

Définition 5.2 (Affectation) Lorsque l'on demande de stocker une valeur dans une variable, on dit que l'on procède à une *affectation*.

Exemple 5.3 *A prend la valeur 7* s'écrit en algorithmique $A \leftarrow 7$ et se traduit en `Python` par `A = 7`.

Une instruction d'affectation comme $y \leftarrow x + 5$ est exécutée de la façon suivante :

- évaluer l'expression $x + 5$ en ajoutant 5 à la valeur contenue dans la variable x,
- mettre le résultat dans la variable y.

Dans une affectation, le membre de gauche reçoit la valeur du membre de droite.

Pour affecter une valeur à une variable, on utilise le symbole =.

- Le nom d'une variable est une succession de caractères alphanumériques ne contenant aucun espace, aucun caractère spécial sauf l'underscore _ et ne commençant jamais par un nombre (exemple : `2im` n'est pas un nom valide, mais `im2` est correct). Il peut contenir des majuscules ou des minuscules. Idéalement le nom d'une variable doit être court et explicite. Attention cependant aux mots clés que l'on ne peut pas utiliser comme nom de variable (par exemple : `int` ne peut pas être un nom de variable). De plus il vaut mieux éviter d'utiliser de caractères accentués.
- La variable affectée prend le type de la valeur que l'on affecte. Par exemple, si l'on affecte `3` à `x`, la variable `x` est de type `int`.

On distingue plusieurs types d'affectations : les affectations simples, les affectations augmentées et les affectations multiples.

5.1 Les affectations simples

Pour pouvoir affecter une valeur à une variable, on utilise la syntaxe suivante :

```
nom_variable = valeur
```

Exemple 5.4 L'instruction `a = 3.5` affecte à la variable `a` la valeur `3.5`. Elle est alors du type `float`, c'est un nombre flottant. L'instruction `a = "abc"` affecte à la variable `a` la valeur `"abc"`. Elle est alors du type `str`, c'est une chaîne de caractères.

5.2 Les affectations augmentées

En `Python` on peut modifier la valeur d'une variable en procédant à une affectation augmentée. Il s'agit simplement d'un raccourci d'écriture, par exemple l'instruction `a = a + 5` permettant d'ajouter 5 à la valeur contenue dans la variable `a` peut s'écrire plus simplement `a += 3`. Attention à ne pas mettre d'espace entre l'opérateur et le symbole d'affectation.

Exemple 5.5

```
>>> a = 5
>>> a *= 3
>>> a
15
```

```
>>> a = 3.1
>>> a -= 1.5
>>> a
1.6
```

```
>>> a = 15
>>> a /= 2
>>> a
7.5
```

```
>>> a = "abc"
>>> a += "d"
>>> a
"abcd"
```

Remarque 5.6 Les opérateurs + et $\star$ n'agissent pas de la même façon selon le type de données qu'ils manipulent. Par exemple, quand il s'agit d'une chaîne de caractère comme dans l'exemple ci-dessus, l'opérateur + concatène les données. Il faut donc avoir conscience du type de données que l'on manipule quand on programme (voir partie 6).

5.3 Les affectations multiples

On peut également procéder à une affectation simultanée de plusieurs variables. Pour se faire, on utilise la syntaxe suivante :

```
nom_variable_1,nom_variable_2,...,nom_variable_n = liste de n valeurs
```

Les n valeurs sont affectées de gauche à droite aux variables déclarées à gauche de l'égalité. La liste de ces n valeurs peut être écrite sous l'une des formes suivantes :

- $\star$ directement et séparées par des virgules : `val_1,val_2,...,val_n`,
- $\star$ avec un tuple : `(val_1,val_2,...,val_n)`,
- $\star$ avec une liste : `[val_1,val_2,...,val_n]`.

Exemple 5.7 Considérons les instructions suivantes :

```
a,b = 1,"essai" # affecte à a la valeur 1 et à b la valeur "essai"
a,b,c = (1,3,-5) # affecte à a la valeur 1, à b la valeur 3 et à c la valeur -5
a,b = ["essai",True] # affecte à a la valeur "essai" et à b la valeur True
```

L'affectation simultanée permet également d'échanger des valeurs facilement. Ainsi, la syntaxe

```
a,b = b,a
```

permet d'échanger les valeurs des variables `a` et `b`.

6 Les types de données

Dans la machine l'information étant binaire, le type des variables permet de la traduire en utilisant le bon « décodeur ».

Définition 6.1 (Types) Les valeurs prises par une variable sont classées par *type*. On utilisera en particulier les nombres entiers positifs, les nombres à « virgule » et les chaînes de caractères.

En `Python` il n'y a pas de déclaration nécessaire : `Python` attribue le type au moment de l'exécution de l'instruction d'affectation.

Exemple 6.2

- $\star$ `A = 17` : A est du type `int` pour *integer* c'est-à-dire un nombre entier ;
- $\star$ `A = 17.0` : A est du type `float` ou nombre flottant c'est-à-dire un nombre décimal ;

- `A = 'bonjour'` : A est du type `str` pour *string* c'est-à-dire une chaîne de caractères ;
- `A = True` : A est du type `bool` c'est-à-dire une donnée booléenne (soit vrai, soit faux) ;
- `A = [1, 2.5, 7, 'a']` : A est du type `list` c'est-à-dire une liste ;
- `A = (1, "abc", True)` : A est du type `tuple` c'est-à-dire un tuple (ou n-uplet).

Les types que nous utiliserons le plus souvent sont les suivants :

nombre entier	nombre flottant	vide	booléen	range	liste	tuple
`int`	`float`	`NoneType`	`bool`	`range`	`list`	`tuple`

Remarque 6.3 Pour les nombres à « virgule », on utilise le point `.` pour indiquer la virgule. On écrit donc `6.4` et non pas `6,4`.

Pour obtenir le type d'une donnée ou d'une variable, on utilise la fonction `type()`.

Exemple 6.4 L'instruction `type(5)` renvoie `<class 'int'>`, l'instruction `type(a)` quand la variable `a` contient la valeur `4.5` renvoie `<class 'float'>`.

Remarque 6.5 Soit n un entier naturel. On accède au n-ième caractère d'une chaîne de caractères ou au n-ième élément d'une liste ou d'un tuple, tous de longueur strictement supérieure à n et contenu dans une variable `A`, avec l'instruction `A[n]`.

6.1 Le transtypage

Pour pouvoir modifier certains types de données ou pouvoir appliquer certaines fonctions ou méthodes, il est parfois nécessaire de passer d'un type à un autre. On procède alors à un *transtypage*. Pour ce faire, il suffit d'appliquer à la donnée que l'on souhaite transtyper la fonction donnant le nouveau type.

Exemple 6.6

- Si `x` est un flottant, l'instruction `int(x)` transtype `x` en un entier (en le tronquant à l'unité). Par exemple, `int(7.2)` renvoie `7`.
- Si `n` est un entier, l'instruction `float(n)` transtype `n` en un flottant. Par exemple, `float(5)` renvoie `5.0`.

Certains transtypages sont toujours valides. Par exemple, l'instruction `str(x)` renvoie toujours une chaîne de caractères, quelque soit le type de la donnée `x`. En revanche, certains transtypages peuvent ne pas fonctionner suivant le type de données utilisées.

Exemple 6.7 L'instruction `int("3")` renvoie `3` et donc permet de transtyper la chaîne de caractères `"3"` en un entier. En revanche, l'instruction `int("3.4")` renvoie une erreur.

6.2 Les opérateurs + et ⋆

Les opérateurs `+` et `⋆` sont compatibles avec la plupart des types mais ne produisent pas le même effet selon le type. Avec des données numériques, on obtient l'addition et la multiplication usuelles. Par contre, avec les chaînes de caractères, les listes et les tuples, l'opérateur + produit une concaténation et l'opérateur ⋆ concatène plusieurs fois les mêmes données.

Exemple 6.8 L'instruction `'abc'+'def'` produit la chaîne de caractères `'abcdef'`.
L'instruction `[1,2]*3` produit la liste `[1,2,1,2,1,2]`.

Attention, les objets manipulés avec les opérateurs + et ⋆ doivent être de types compatibles.

Exemple 6.9 Les instructions `1+'abc'` et `[1,2,3]+(4,5,6)` produisent une erreur.
Par contre l'instruction `7.4+3` ne produit pas d'erreur, les types `int` et `float` sont compatibles pour l'addition.

7 L'éditeur de scripts

L'éditeur de scripts permet d'écrire plusieurs instructions que l'on peut ensuite faire interpréter en une seule fois (dans `EduPython` on clique sur la flèche verte située en haut, au milieu de la barre d'outils). Il permet également d'écrire des programmes beaucoup plus longs et plus évolués. Il facilite notamment la programmation par son système de coloration syntaxique mettant en avant les mots clés du langage, soulignant des erreurs de syntaxe, et indentant automatiquement.

7.1 Encodage

Pour assurer la compatibilité entre ordinateurs, il faut veiller à commencer ses scripts par :

```
# -*- coding: utf-8 -*-
```

Nous verrons plus tard dans l'année ce qu'est l'encodage `utf-8` qui permet ici de s'assurer notamment que les accents sont bien codés de la même façon et sont donc lisibles.

7.2 Les commentaires

Les commentaires sont des instructions qui ne sont pas interprétées lors de l'exécution du programme. Ils permettent d'expliciter certaines instructions, astuces ou parties du programme (documentation de fonction par exemple) et sont nécessaires au maintien du code par un groupe de programmeurs.

- Lorsque les commentaires sont situés sur une seule ligne, on les fait précéder du symbole # :

```
# on tape les commentaires sur une seule ligne
```

- Lorsque les commentaires sont situés sur plusieurs lignes, comme lors de la documentation d'une fonction, on les encadre par une série de trois guillemets :

```
""" on tape les commentaires sur plusieurs lignes """
```

8 Les fonctions

En informatique, une fonction est une suite d'instructions (sorte de sous-programme) que l'on peut appeler au besoin. On définit une fonction pour éviter les répétitions, ou pour décomposer un programme complexe en sous-tâches plus simples. Les fonctions permettent de rendre un programme plus lisible.

Définition 8.1 Une *fonction* est définie par un *nom* et correspond à une séquence d'instructions réalisant une tâche précise, en utilisant un ou plusieurs *arguments*, voire aucun. Ces arguments sont les *paramètres* de la fonction. La fonction renvoie un résultat.

Utiliser des fonctions permet de découper un problème en sous-problèmes et d'éviter ainsi la répétition fastidieuse d'instructions.

8.1 Définir une fonction

Pour définir des *fonctions* en `Python` on utilise le mot-clé **`def`**. La syntaxe pour la création d'une fonction est la suivante :

```
def nom_fonction(liste_paramètres) :        # ne pas oublier les :
   """ documentation de la fonction """     # chaîne de caractères entre triple guillemets
   bloc d'instructions                      # indentation obligatoire
   return résultat                          # renvoie le résultat de la fonction
# pour terminer, on stoppe l'indentation
```

Nota Bene :

- la liste des paramètres peut être vide ;
- la documentation de la fonction est très importante. Elle fournit des informations sur les types des paramètres, sur ce qu'elle fait et sur ce qu'elle renvoie. On accède à cette documentation en utilisant l'instruction `help(nom_fonction)` ;
- l'instruction `return` stoppe l'exécution de la fonction et renvoie une ou plusieurs valeurs. De cette façon on peut récupérer le résultat renvoyé par une fonction. Si plusieurs valeurs sont renvoyées, elles le sont sous forme d'un tuple ;
- si l'instruction `return` est omise, la fonction renvoie `None` et on parle alors de *procédure*.

On voir la notion de fonction comme étant un mini-algorithme puisque les paramètres sont les données d'entrée, les instructions composant le corps de la fonction (partie indentée) correspondent au traitement des données, les valeurs renvoyées sont les données de sortie.

Exemple 8.2 La fonction `distance_parcourue` ci-dessous prend en argument la vitesse en km/h et le temps en secondes et renvoie la distance parcourue en mètres :

Algorithme

```
fonction distance_parcourue(vitesse,temps)
   vitesse_ms ← vitesse*1000/3600
   distance ← vitesse_ms * temps
   renvoyer distance
fin fonction
```

Programmation `Python`

```
def distance_parcourue(vitesse,temps):
   vitesse_ms = vitesse*1000/3600
   distance = vitesse_ms * temps
   return distance
```

Une fois une fonction définie, elle peut être « appelée » tout au long de l'exécution de l'algorithme (ou du programme), autant de fois que nécessaire.

8.2 Appel d'une fonction

Pour utiliser une fonction, il faut l'appeler et lui passer des valeurs par arguments si elle possède des paramètres. Les valeurs passées doivent bien sûr être dans le même ordre que les paramètres et du même type que ceux-ci.

Exemple 8.3 Pour connaître alors la distance parcourue par un véhicule roulant à 72 km/h pendant 2 minutes, on saisit l'instruction

```
distance_parcourue(72,120)
```

qui nous renvoie la valeur `2400.0`.
On dit que l'on procède à *l'appel de la fonction* `distance_parcourue` pour les valeurs 72 et 120.

Les paramètres d'une fonction peuvent être eux-mêmes des fonctions.

Exemple 8.4 Considérons les trois fonctions suivantes :

```
def cube(x):
    """
    Fonction cube
    """
    return x**3
```

```
def monAffine(x):
    """
    Fonction affine
    f(x) = 2x + 3
    """
    return 2*x + 3
```

```
def composition(f,g,x):
    """
    Renvoie f(g(x))
    """
    return f(g(x))
```

La troisième fonction prend en paramètres deux fonctions f et g. En tapant l'instruction `composition(cube,monAffine,1)` on obtient ainsi le cube de 5 soit 125. En tapant `composition(monAffine,cube,2)` on calcule cette fois l'image du cube de 2 (soit 8) par la fonction affine $x \mapsto 3x + 2$ et on obtient 19. La fonction `composition` est réutilisable avec d'autres fonctions compatibles c'est-à-dire telles que l'ensemble image de g est inclus dans l'ensemble de définition de f, par exemple on ne peut pas calculer la racine carrée d'un nombre négatif, ni calculer l'inverse de zéro.

8.3 Portée des variables : variables locales et variables globales

Lorsqu'une fonction est appelée, `Python` réserve pour elle (dans la mémoire de l'ordinateur) un espace de noms. Cet espace de noms local à la fonction est différent de l'espace de noms global où se trouvent les variables du programme principal. Dans l'espace de noms local, nous aurons des variables qui ne sont accessibles qu'au sein de la fonction.

- Les variables qui sont définies dans le corps d'une fonction sont des variables *locales*, c'est-à-dire qu'elles n'existent qu'à l'intérieur de cette fonction. En particulier, une fois la fonction exécutée, ces variables sont détruites et n'existent plus. Une variable locale peut avoir le même nom qu'une variable de l'espace de noms global mais elle reste néanmoins indépendante.
- Les variables qui sont définies en dehors du corps de toutes fonctions sont appelées variables *globales*. En `Python` ces variables sont accessibles à l'intérieur du corps d'une fonction uniquement en *lecture*, autrement dit elles leur contenu est « visible », mais la fonction ne peut pas le modifier sauf si on les redéclare explicitement avec le mot-clé `global`.

Exemple 8.5 Une fois l'instruction `distance_parcourue(72,120)` exécutée (voir exemples 8.2 et 8.3) les variables `vitesse_ms` et `distance` n'existent plus, ce sont des variables locales à la fonction `distance_parcourue`.

Exemple 8.6 Les instructions ci-contre renvoient une erreur du type `UnboundLocalError`, c'est-à-dire que l'on a essayé de modifier une variable non définie dans la fonction `modif()`.

Pour pouvoir modifier une variable globale à l'intérieur d'une fonction (et donc modifier globalement sa valeur), il faut utiliser le mot-clef **`global`**. Dans l'exemple précédent, en ajoutant l'instruction `global a` juste avant de modifier la valeur de la variable `a` on peut modifier la valeur de `a`.

```
a = 3

def modif():
    a += 1
    return a

modif()
```

```
a = 3

def modif():
    global a
    a += 1
    return a

modif() ; a
```