

Informatique et Algorithmique avec le langage Python

Cours

Sabine MARDUEL, révisions Laurent POINTAL

DUT Mesures Physiques - semestre 2

Table des matières

Programme du module.....	3
I - Algorithmes, instructions et langages informatiques.....	5
1) L'algorithmique.....	5
2) Langages de programmation.....	6
a) Langages machine.....	6
b) Langages évolués : de "haut niveau".....	6
3) Le langage python.....	6
a) Utilisation de python.....	6
a.1) L'interpréteur python appelé aussi Shell.....	7
a.2) L'éditeur de programmes (ou éditeur de scripts).....	7
a.3) Installation de python sur votre ordinateur personnel.....	8
4) Les instructions.....	10
a) Instructions simples.....	10
b) Instructions composées.....	10
b.1) Instruction conditionnelle si.....	10
b.2) Instruction de boucle pour.....	11
b.3) Instruction de boucle tant que.....	11
II - Variables, données et opérateurs.....	13
1) Variables et affectation de données.....	13
2) Nommage des variables.....	14
a) Les constantes.....	15
3) Les types de données.....	15
a) Expression littérale des données.....	16
4) Déclaration et initialisation d'une variable.....	16
5) Le transtypage ou conversion de type.....	17
6) Les opérateurs.....	17
III - Les programmes python - les modules.....	19
1) Exemples.....	19
a) Exemple 1 - calculs.....	19
b) Exemple 2 - graphisme.....	19
2) Suivre l'exécution d'un programme : le tableau de suivi.....	19
3) Tester un programme : le jeu d'essais.....	20
4) Présenter un module : en-tête, commentaires, résultats d'exécution.....	20
IV - Les fonctions : utilisation.....	23
1) Importation de fonctions prédéfinies depuis des "bibliothèques".....	23
2) Fonctions d'aide.....	24
3) Fonctions d'entrée et sortie.....	24
4) Appeler une fonction prédéfinie.....	25
5) Valeur retournée par une fonction.....	25
V - Les booléens et l'instruction conditionnelle if.....	27
1) Le type booléen.....	27
a) Opérateurs de comparaison.....	27
b) Opérateurs sur les booléens.....	27
c) Tables de vérité.....	28
c.1) Opérateur NON (not en python).....	28
c.2) Opérateur ET (and en python).....	28
c.3) Opérateur OU (or en python).....	28
2) Algèbre de Boole et logique booléenne.....	28
a) Opérateurs NON, ET, OU.....	29
b) Propriétés.....	29
b.1) Propriétés de NON.....	29
b.2) Propriétés de ET.....	29
b.3) Propriétés de OU.....	29
b.4) Distributivité.....	29
b.5) Lois de De Morgan.....	29
c) Autres portes logiques.....	30
3) L'instruction conditionnelle if.....	30
VI - Les séquences - l'instruction de boucle for.....	31
1) Boucle for.....	31
a) Syntaxe.....	31
2) Séquences ou types itérables.....	31
a) Générateur range.....	32
b) Les listes.....	32
b.1) Opérations et syntaxes de base.....	32
b.2) Listes en compréhension.....	35
c) Les chaînes de caractères.....	36
d) Le tuple.....	37
e) Autres types itérables.....	37
e.1) Dictionnaire.....	37
e.2) Ensemble.....	38
e.3) Erreurs.....	38
VII - L'instruction de boucle while.....	39
1) Principe.....	39
2) Difficultés.....	40
a) Éviter les résultats faux à la sortie de la boucle.....	41
b) Éviter les boucles infinies.....	41
b.1) Conseils & astuces.....	42
3) Boucle while ou boucle for ?.....	43
VIII - Les fonctions : créer des sous-programmes.....	45
1) Définir une fonction.....	45
2) Exemples.....	45
3) Le programme principal.....	46
4) Paramètres d'une fonction.....	48
5) Appeler une fonction.....	48
a) Les arguments remplacent les paramètres.....	48
b) Ordre des arguments.....	49
c) Paramètres "optionnels" avec une valeur par défaut.....	49
6) Portée des variables.....	49
7) Fonctions récursives.....	50
8) Méthodes.....	51
IX - Organisation des applications en plusieurs modules.....	53
a) Définition d'un module.....	53
b) Utilisation d'un module.....	53
c) Renommage de noms importés.....	54
X - Les fichiers.....	55
1) Ouvrir un fichier.....	55
2) Fermer un fichier.....	56
3) Ecrire dans un fichier.....	56
4) Lire un fichier.....	57
a) Boucle de lecture.....	58
b) Lecture avec bloc gardé.....	58
5) Organisation des fichiers sur disque.....	59
a) Nommage.....	59
b) Arborescence.....	59
c) Séparateur de noms dans l'arborescence.....	59
d) Notion de répertoire courant.....	59
e) Chemins de fichiers.....	60
XI - Les exceptions.....	61
1) Capture des exceptions.....	61
2) Traitement des exceptions.....	61

3) Signalisation d'erreur : levée d'exception.....	63	1) Webographie.....	71
XII - Programmation Orientée Objet.....	65	2) Bibliographie.....	71
1) Utilisation des objets.....	65	3) Codes hexadécimaux.....	72
2) Création de familles d'objets... les classes.....	65	4) Priorité des opérateurs.....	72
XIII - Le module matplotlib.....	69	5) Récapitulatif des opérateurs booléens.....	73
1) Utilisation.....	69	6) Différences entre Python 2 et Python 3.....	73
XIV - Annexes.....	71		

Programme du module

Programme pédagogique national (PPN) du DUT Mesures Physiques :

UE 12	Outils de la mesure	45h (7h CM, 10h TD, 28h TP)
M 1204	Algorithmique et informatique	Semestre 1
Objectifs du module : Approfondir les compétences en algorithmique. Définir la structure d'un programme et élaborer un utilitaire.		
Compétences visées : Concevoir et mettre en œuvre un algorithme. Mettre en œuvre un traitement numérique de données expérimentales.		
Prérequis : Programme du lycée des séries S, STI2D, STL.		
Contenus : Tests logiques, conditionnelles imbriquées, boucles imbriquées. Fonctions et procédures. Tableaux à une et à deux dimensions. Modularité, gestion des entrées/sorties. Test d'un programme. Principes de la programmation orientée objet.		
Modalités de mise en œuvre : Travaux pratiques sur ordinateur, principalement avec un langage textuel. Documentation des programmes et commentaires. Traitement de fichiers et de données. « Apprendre autrement » : informatisation de calculs de physique ou de chimie, de métrologie, ou d'analyse numérique à l'aide d'un langage informatique.		
Prolongements possibles : Réinvestissement de l'outil informatique dans les autres modules scientifiques.		
Mots clés : Algorithme, programmation, modularité, langage, entrées-sorties.		

I - Algorithmes, instructions et langages informatiques

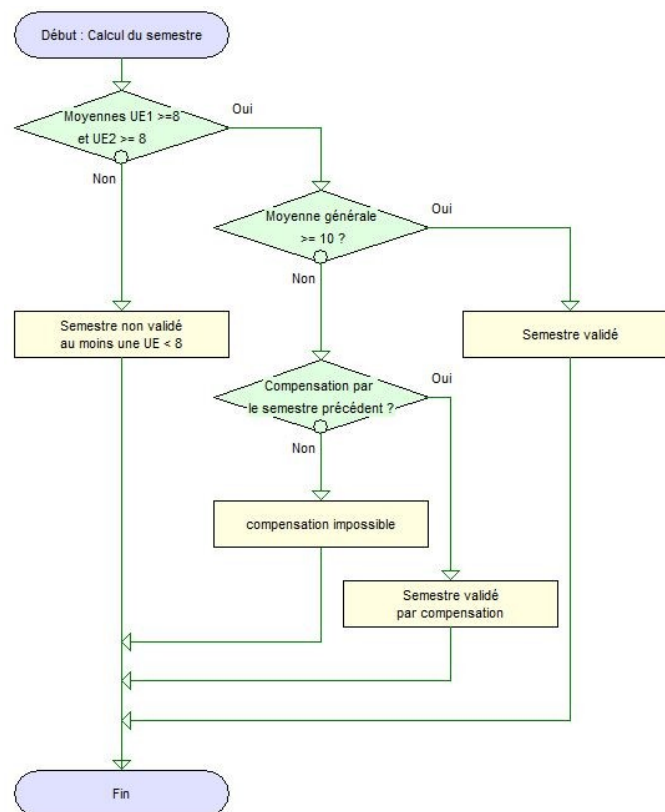
1) L'algorithmique

Un **algorithme** est une suite finie d'instructions, écrites en langage naturel, qui peuvent être exécutées les unes à la suite des autres pour résoudre un problème.

L'algorithme ne dépend pas du langage de programmation dans lequel il sera traduit, ni de la machine qui exécutera le programme. Exemples d'algorithmes mis en œuvre « naturellement » tous les jours : recette de cuisine, notice de montage d'un appareil, tri de nombres par ordre croissant, recherche dans un annuaire ; et d'algorithmes déjà vu en cours : calcul de la factorielle d'un nombre entier, résolution d'une équation du second degré...

Un algorithme peut aussi être représenté sous forme graphique, on parle d'**organigramme** (ou d'ordino-gramme).

Exemple d'organigramme : validation d'un semestre



Un **programme informatique** (appelé aussi “application”) est une traduction de l'algorithme dans un *langage de programmation*.

L'ordinateur peut alors exécuter le programme pour obtenir le résultat voulu. Il exécute les instructions de l'algorithme les unes à la suite des autres.

2) Langages de programmation

a) Langages machine

Les microprocesseurs des ordinateurs travaillent sur des données binaires 0/1, appelées des **bits**, que l'on regroupe par **octets** (groupe de 8 bits). Chaque famille de processeurs comprend un jeu d'instructions (d'opérations qu'il peut réaliser) avec une représentation binaire propre.

Exemple en langage binaire x86 d'un programme calculant et 4+5

```
01010101 10001001 11100101 10000011 11101100 00010000 11000111 01000101 11110100
00000100 00000000 00000000 00000000 11000111 01000101 11111000 00000101 00000000
00000000 00000000 10001011 01000101 11111000 10001011 01010101 11110100 10001101
00000100 00000010 10001001 01000101 11111100 10001011 01000101 11111100 11001001
11000011
```

Comme le binaire est difficilement lisible, on utilise très souvent la **représentation hexadécimale** (un chiffre de 0 à f représente 4 bits - voir Codes hexadécimaux page 72), voici la représentation du même programme sous cette forme :

```
55 89 e5 83 ec 10 c7 45 f4 04 00 00 00 c7 45 f8 05 00 00 00 8b 45 f8 8b 55 f4 8d 04 02
89 45 fc 8b 45 fc c9 c3
```

b) Langages évolués : de “haut niveau”

De très nombreux langages informatiques existent¹. Les langages « assembleurs » sont ceux qui sont les plus proches de ce que comprennent les processeurs, ils représentent les opérations exécutables et leurs options sous forme textuelle compréhensible (pour peu qu'on en apprenne le sens).

Voici quelques exemples de langages de programmation (dont vous avez probablement déjà entendu parler) :

Nom du langage	Apparu en
Assembleur	1950
Basic	1964
Pascal	1970
C	1973
C++	1983
Python	1991
Java	1994

3) Le langage python

C'est un langage objet, de nouvelle génération, pseudo-interprété, portable. Il est libre, ouvert, gratuit. De nombreuses “bibliothèques” sont disponibles sur internet. L'auteur de ce langage est *Guido van Rossum*.

Son aspect proche de l'algorithmique fait qu'il a été choisi en de nombreux endroits pour l'enseignement. Mais Python est aussi utilisé dans le « monde réel » pour des applications : moteur de recherche Google, Youtube, laboratoires de recherche (CNRS, INRIA, Universités...), agences spatiales (NASA...), jeux vidéo, cinéma, finance, etc. Il est entre autres utilisé dans de nombreuses entreprises pour de l'informatique d'instrumentation (collecte et analyse de données).

a) Utilisation de python

On peut utiliser python depuis une **fenêtre de terminal** (ou *console*) ou bien, on peut passer par un **environnement de développement** (IDE - Interactive Development Environment) c'est à dire un éditeur de texte muni de différentes fonctions pour faciliter la programmation.

Nous utiliserons principalement l'environnement de développement **Pyzo** (mais il existe beaucoup d'autres environnements de développement comme IPython/Jupyter, WingIDE, PyCharm, etc). Précédemment nous utilisions IDLE, qui a l'avantage d'être installé par défaut avec Python sous Windows (sous Linux il faut généralement installer un paquet logiciel complémentaire, et sous MacOS X c'est plus compliqué²).

Dans tous les cas, il y a deux façons de travailler avec python : l'*interpréteur* et l'*éditeur*.

1 - Voir par exemple https://fr.wikipedia.org/wiki/Liste_de_langages_de_programmation

2 - Voir les indications sur <https://www.python.org/download/mac/tcltk/>

a.1) L'interpréteur python appelé aussi Shell.

Il permet de saisir des instructions qui sont immédiatement exécutées, comme sur une calculatrice.

Depuis une **fenêtre de terminal**³ : on lance le Shell Python en saisissant **python** (sous Windows) ou bien **python3** (sous Linux ou MacOS X). Une fois dans le Shell Python, on en sort en écrivant **exit()** (ou via les raccourcis clavier Ctrl-D sous Linux/MacOS X, ou Ctrl-Z+enter sous Windows).

Dans l'**environnement de développement Pyzo** le Shell Python et l'éditeur apparaissent dans la même fenêtre, le Shell Python se trouve dans une des zones d'outils de la fenêtre (ces zones sont réorganisables suivant le choix de l'utilisateur).

Dans l'**environnement de développement IDLE** on arrive directement dans la fenêtre du Shell Python.

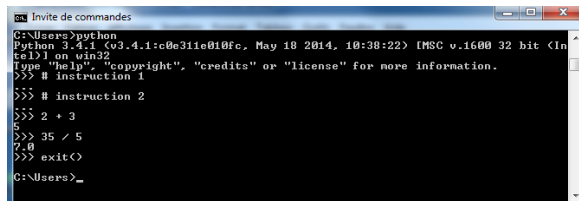


Illustration 1: Fenêtre de terminal console

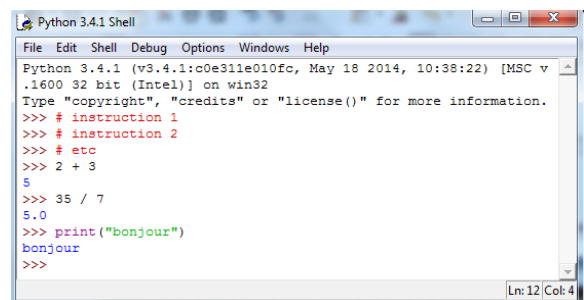


Illustration 2: Environnement de développement IDLE

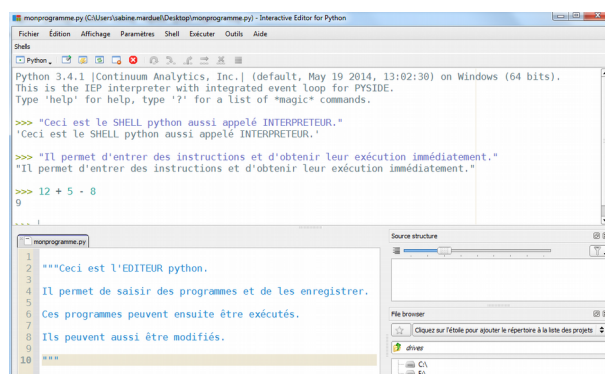


Illustration 3: Environnement de développement Pyzo

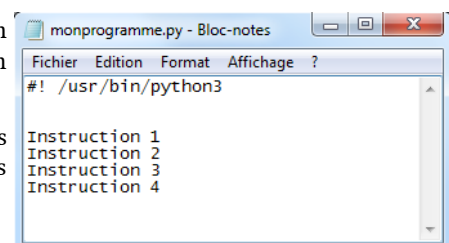
Lors d'utilisation du Shell Python, un "prompt"⁴ `>>>` est affiché par l'environnement et invite à la saisie d'une commande, lors de l'appui sur « Entrée » la commande est évaluée et le résultat de cette évaluation affiché⁵ :

```
>>> x = 4
>>> x
4
```

a.2) L'éditeur de programmes (ou éditeur de scripts)

Le programme (aussi appelé *script*) devra être enregistré dans un **fichier texte d'extension .py** pour qu'il soit reconnu comme un programme python.

On peut écrire ce programme dans un simple éditeur de texte⁶ : dans l'exemple ci-contre, sous Windows, le programme est enregistré dans un fichier nommé **monprogramme.py**.



3 - Sous Windows une fenêtre de terminal s'obtient en ouvrant l'application *Invite de commandes* (ou en demandant de lancer l'exécutable `cmd.exe`). Sous Linux, ouvrir *Gnome Terminal* ou *Konsole* ou.... Sous MacOS X, ouvrir *Applications* → *Outils* → *Terminal*.

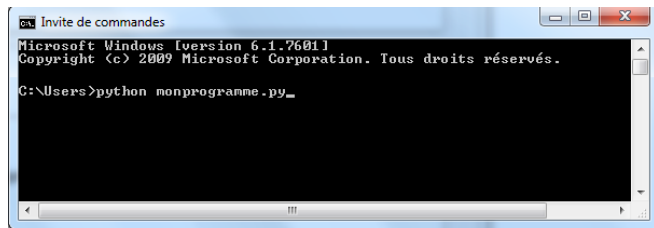
4 - Ou "invite de commande".

5 - Si commande produit un résultat.

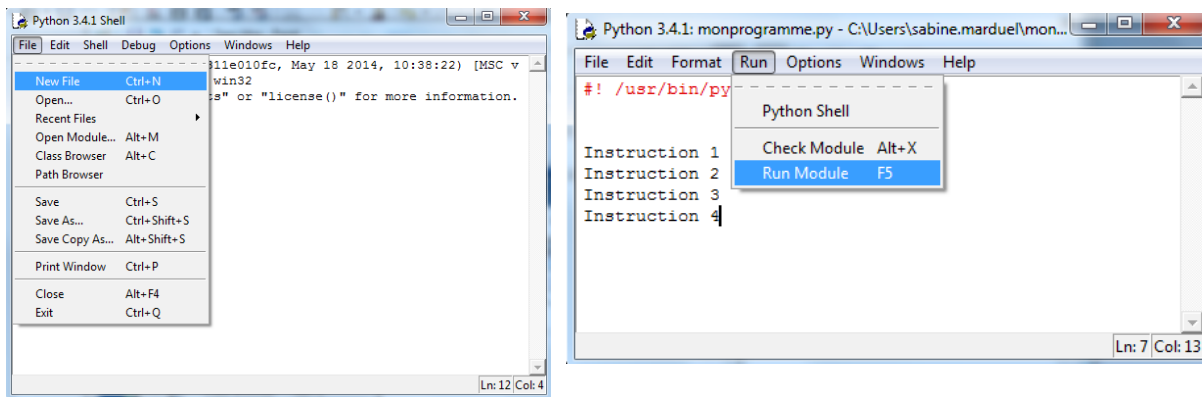
6 - Notepad, Notepad++, gEdit, Kate, TextWrangler, SublimeText... Vim, Emacs... suivant votre environnement.

Après l'avoir écrit et enregistré, on l'exécute depuis une fenêtre de terminal⁷, en se déplaçant dans le répertoire qui contient le script (cd ...) et en saisissant le mot **python** (sous Windows) ou **python3** (sous Linux ou MacOS X) suivi du nom du fichier script programme. Dans notre exemple, sous Windows, on écrit :

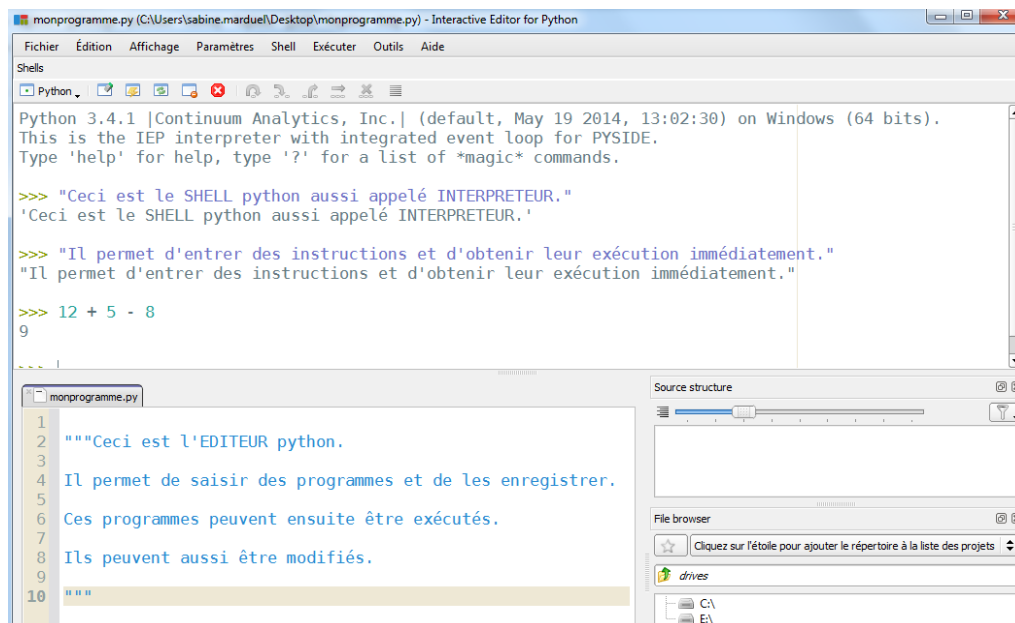
```
python monprogramme.py
```



Ou bien, si on utilise l'environnement de développement IDLE, on écrit le programme en créant un nouveau fichier (menu *File* → *New File*), on l'enregistre sous le nom de notre choix, puis on l'exécute en cliquant le menu *Run* → *Run module* :



Ou encore, dans l'environnement de développement Pyzo, l'éditeur apparaît dans une zone de la fenêtre :



a.3) Installation de python sur votre ordinateur personnel

Si vous avez un ordinateur personnel (celui-ci n'a pas besoin d'être puissant), vous pouvez installer **Python3** soit avec l'environnement de développement **IDLE3**, soit avec **Pyzo** (environnement utilisé à l'IUT pour les TPs), comme indiqué ci-après.

Si vous ne disposez pas d'un ordinateur chez vous, vous pourrez venir faire votre travail personnel à l'IUT dans la salle « Chablis » (bâtiment A, salle A017).

⁷ - Sous Windows, il faut avoir coché l'option d'ajout de Python dans le PATH lors de l'installation.

Pyzo

Pour installer Pyzo, rendez-vous à l'adresse web <http://www.pyzo.org/start.html> et suivez les instructions correspondant à votre plateforme (Windows / Linux / MacOS X). Les développeurs de Pyzo conseillent d'installer un environnement Python *Anaconda* ou *Miniconda* (version moins volumineuse de Anaconda), ceci permet d'installer directement des packages scientifiques de base (calcul, graphiques) et facilite l'installation d'autres packages. Suivez les autres étapes pour configurer l'environnement.

IDLE3

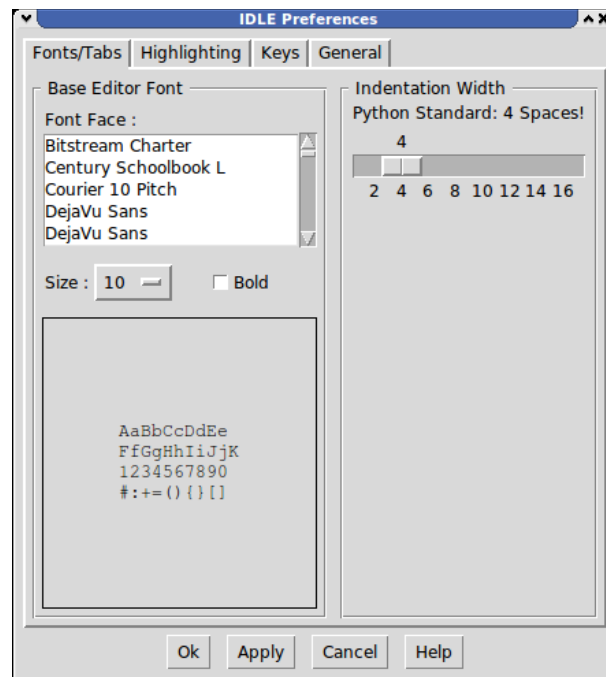
Si vous avez un **PC sous Linux** (Ubuntu, Debian ou autre) : Python est probablement déjà installé. Si vous avez déjà Python3 (essayez de lancer `python3` dans une console), laissez-le tel quel (peu importe le numéro secondaire de version (3.2, 3.3, 3.4...)). Si vous avez Python2 mais pas Python3, n'enlevez surtout pas Python2 (car il est nécessaire pour le bon fonctionnement de votre système) et rajoutez Python3 en l'installant depuis le gestionnaire de paquets ; en plus de Python3 choisissez le paquet qui offre IDLE3 (rajoutez aussi Tkinter si c'est un paquet optionnel).

Si vous avez un **Mac** : Python est probablement déjà installé. S'il s'agit de `python2`, ne l'enlevez surtout pas car il est utile pour le fonctionnement du système d'exploitation. Vous pouvez installer `python3` sans enlever `python2` en suivant les instructions Windows de ci-dessous.

Si vous avez un **PC sous Windows** (ou un Mac) où Python3 n'est pas présent :

- 1) Téléchargez l'installateur Python3 à l'adresse suivante :
 - Pour Windows : <https://www.python.org/downloads/windows/>
 - Pour MacOS X : <https://www.python.org/downloads/mac-osx/>
- 2) Choisissez la dernière version « stable » de Python3 (pas une version « rc ») adaptée à votre système (32/64 bits).
- 3) Lancez l'exécution de l'installateur, Lors de l'installation, si les options sont disponibles validez l'installation de Tkinter, IDLE3 ainsi que l'ajout de python dans le PATH.

Pour configurer l'environnement de développement IDLE, démarrez-le (sous Windows il est listé dans les applications du groupe Python) : menu *Options* → *Configure IDLE*, puis vérifiez que la rubrique **Indentation Width** est bien réglée sur **4 espaces** (4 spaces).



4) Les instructions

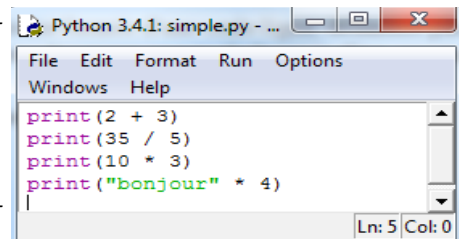
On distingue les instructions simples et les instructions composées.

a) Instructions simples

En python, une instruction simple est formée d'une seule ligne, délimitée par le caractère invisible de fin de ligne⁸. Exemples d'instructions simples (dans l'interpréteur python) :

```
>>> 2 + 3
>>> 35 / 5
>>> 10 * 3
>>> "bonjour" * 4
```

Ci-contre un exemple de **programme** python contenant des instructions simples similaires.



👉 Un caractère # placé sur la ligne d'une instruction simple introduit un commentaire jusqu'à la fin de la ligne. Ce commentaire est à destination des programmeurs, il est ignoré par Python.

b) Instructions composées

En python, la structuration des **blocs d'instructions** se fait grâce à l'**indentation**⁹ (le décalage visuel avec des espaces) : les lignes consécutives qui ont la même indentation appartiennent au même bloc ; une ligne ayant une indentation moindre (ou la fin du programme) termine le bloc d'instructions constitué par les lignes qui la précèdent.

Une **instruction composée** est formée d'une **instruction d'introduction** terminée par le caractère deux-points (:), suivi par un **bloc d'instructions** simples (ou elles-mêmes structurées) indentées par rapport à cette instruction d'introduction.

instruction d'introduction :	(ligne terminée par un caractère deux-points)
instruction 1	(bloc d'instructions secondaire,
instruction 2	composé de une ou plusieurs
instruction 3	lignes d'instructions)
autre instruction	(autre instruction hors du bloc, entraînant la fin du bloc précédent)

Les lignes du bloc secondaire d'instructions sont alignées entre elles et décalées (indentées) par rapport à la ligne d'introduction (convention de décalage de 4 espaces).

N'utilisez pas de caractère tabulation pour l'indentation des scripts Python, réglez votre éditeur pour que l'appui sur la touche tabulation génère une indentation avec 4 espaces.

Les principales instructions composées sont l'instruction conditionnelle **si** (**if** en python), l'instruction de boucle **pour** (**for** en python), et l'instruction de boucle conditionnelle **tant que** (**while** en python).

👉 Lorsqu'une instruction ouvre une expression avec une parenthèse (ou une accolade { ou un crochet [, alors **l'indentation est ignorée** jusqu'à ce que le) ou } ou] correspondant referme l'expression.

b.1) Instruction conditionnelle si

Voir chapitre V (Les booléens et l'instruction conditionnelle if) en page 27.

Cette instruction permet d'exécuter un bloc secondaire si une condition est vraie, c'est à dire si un prérequis est réalisé. En algorithmique cette instruction s'écrit de la façon suivante :

-
- 8 - Dans d'autres langages comme le langage C++, une instruction simple se termine par un ;
 - 9 - D'autres langages utilisent d'autres méthodes pour structurer les blocs d'instructions : en langage C, C++, Java, PHP... la structuration se fait grâce à des accolades {}. En langage Pascal, ADA... la structuration se fait grâce aux mots clés begin et end .

```

si condition1 alors :
    instruction 1 (ou bloc d'instructions 1)
sinon :
    instruction 2 (ou bloc d'instructions 2)
fin si

```

On peut imbriquer plusieurs instructions conditionnelles.

Exemple : résultat d'un semestre

```

si (moyenneUE1 ≥ 8 et moyenneUE2 ≥ 8) alors:
    si moyenneGenerale ≥ 10 alors:
        afficher(«Le semestre est validé!»)
    sinon:
        si (moyenneGenerale + moyenneSemestrePrecedent)/2 ≥ 10 alors:
            afficher(«Le semestre est validé par compensation avec le
                semestre précédent.»)
        sinon:
            afficher(«La compensation est impossible. Le semestre n'est
                pas validé.»)
        fin si
    fin si
sinon:
    afficher(«Le semestre n'est pas validé car au moins l'une des UE est
        inférieure à 8.»)
fin si

```

En langage python, cette instruction s'utilise ainsi :

```

if condition1 :
    bloc d'instructions 1
elif condition2 :
    bloc d'instructions 2
elif condition3 :
    bloc d'instructions 3
else :
    bloc d'instructions 4

```

Le mot clé **if** signifie « si », le mot clé **elif** signifie « sinon si » et le mot clé **else** signifie « sinon ».

b.2) Instruction de boucle pour

Voir chapitre VI (Les séquences - l'instruction de boucle for) en page 31.

Cette instruction permet d'exécuter un bloc secondaire plusieurs fois de suite.

Exemple : (en algorithmique)

```

pour n entre 1 et 10 :
    u = n*n
    afficher(« le carré de », n, « est », u)
fin pour

```

b.3) Instruction de boucle tant que

Voir chapitre VII (L'instruction de boucle while) en page 39.

Elle permet d'exécuter un bloc secondaire tant qu'une certaine condition reste vraie.

Exemple : (en algorithmique)

```

    afficher(« entrez un nombre strictement positif »)
    saisir(n)
    tant que n < 0 :
        afficher(« erreur, le nombre n'était pas strictement positif. Recommencez svp. »)
        saisir(n)
    fin tant que

```


II - Variables, données et opérateurs

1) Variables et affectation de données

Les instructions d'un algorithme mettent en jeu des **données** (numériques, texte, etc) qui peuvent être saisies par l'utilisateur, résulter d'un calcul de l'ordinateur, etc. Il est pratique d'enregistrer ces données dans la **mémoire**¹⁰ de l'ordinateur pour pouvoir les réutiliser au fur et à mesure du programme. Les espaces mémoire où ces données sont conservées sont appelés **variables**.

Le rangement d'une donnée (un contenu) dans une variable (un contenant) s'appelle l'**affectation**. Il s'agit de créer une liaison (provisoire) entre un **nom de variable** (aussi appelé *identificateur*) et l'emplacement de la mémoire où est stockée la donnée.

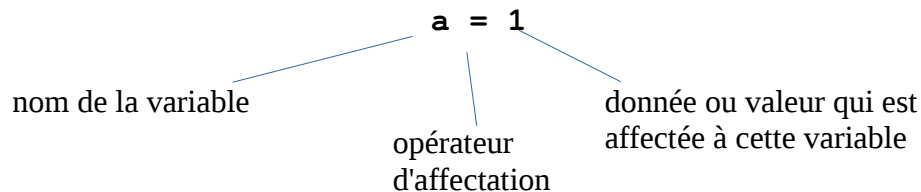
En langage algorithmique, on utilise le symbole \leftarrow pour indiquer qu'une valeur est affectée à une variable :

$$a \leftarrow 1$$

affectation de la valeur 1 à la variable a

la variable a reçoit la valeur 1

En python, on utilise le symbole = qui s'appelle l'opérateur d'affectation (on parle aussi de « name binding »).



Le « nom » (ou identificateur) de la variable peut être vu comme une étiquette attachée à un objet.

Ici, l'objet (un nombre entier) 1 a une étiquette appelée a.



Si on réaffecte "a", on déplace l'étiquette sur un autre objet :



Si l'on affecte un nom à un autre nom, on attache une nouvelle étiquette à un objet existant :



Exemples : saisir ce qui suit dans le Shell Python :

```
>>> a = 1
>>> a
1
>>> print(a)
1
>>> a = 2
>>> a
2
>>> x = 15
>>> x
15
>>> print(x)
15
```

¹⁰ - Mémoire vive ou RAM (Random Access Memory)

☞ Remarque : ne pas confondre l'*opérateur d'affectation* = avec le *symbole d'égalité mathématique*. L'affectation n'est ni une équation mathématique, ni une égalité mathématique !

2) Nommage des variables

En partie par obligation liée au langage, et en partie par convention pour l'enseignement, **un nom de variable doit** :

- débuter par une lettre minuscule sans accent,
- ne contenir que des lettres sans accents, des chiffres et le tiret de soulignement _ (appelé tiret “underscore” ou encore “tiret du 8”),
- être aussi explicite que possible — ne pas utiliser de noms trop courts et sans signification évidente comme `v1`, `v2`, `v3`, `v4`, mais utiliser plutôt `x`, `y`, `a`, `b` (s'ils ont un sens dans le contexte) ou des noms plus explicites comme `age`, `longueur`, `nombre`, `nbr`, `somme`....)

Exemples :

```
>>> variable1 = 30
>>> variable1
30
>>> variable = 8.4
>>> variable
8.4
>>> variable2
NameError: name 'variable2' is not defined
>>> variable1
30
>>> VARIaBle1
NameError: name 'VARIaBle1' is not defined
>>> vâriable
NameError: name 'vâriable' is not defined
```

☞ Le langage Python fait la distinction entre les majuscules et les minuscules (distinction de *casse*), de même qu'entre les caractères avec et sans accent.

Les **mots clés réservés du langage** ne peuvent pas être utilisés comme nom de variables (car ils sont déjà “réservés” par le langage). Ce sont :

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

Exemple :

```
>>> while = 4
SyntaxError: invalid syntax
>>> class = 100
SyntaxError: invalid syntax
```

Attention : il est dangereux et donc déconseillé d'utiliser pour une variable un nom courant python déjà existant (même si ce n'est pas un mot clé réservé). Par exemple :

```
>>> x=4
>>> print(x)
4
>>> print("coucou")
coucou
>>> print
<built-in function print>
>>> print = 18
>>> print("coucou")
TypeError: 'int' object is not callable
>>> print
18
```



```
>>> int = 2
>>> int(3.4)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'int' object is not callable
```

On peut **effacer un nom de variable** avec l'instruction **del**. Par exemple :

```
>>> x = 3
>>> x
3
>>> del(x)
>>> x
NameError: name 'x' is not defined
```

a) Les constantes

Une **constante** est une variable dont la valeur ne doit pas changer au cours de l'exécution du programme. Par convention, on la nomme en MAJUSCULES.

Exemple :

```
NB_MAX_TP = 14      # nombre maximum d'étudiants dans un groupe TP
NOTE_MIN_UE = 8     # note minimale à obtenir pour valider une UE
```

👉 Contrairement à d'autres langages, Python n'empêche pas la modification d'une constante. L'utilisation de la convention avec des majuscules indique aux développeurs qu'il ne faut pas modifier la variable concernée.

3) Les types de données

Chaque donnée et par conséquent chaque variable possède un certain type. Il est important pour l'ordinateur de savoir de quel type est une variable, car cela correspond à une certaine place à réserver en mémoire.

nom en français	nom python	nom en anglais	exemple
nombre entier	int	integer	4
nombre flottant (décimal, réel)	float	float	10.25
chaîne de caractères ¹¹	str	string	"bonjour tout le monde"
booléen	bool	boolean	True (vrai) False (faux)
complexe	complex	complex	1j 2+3j

L'instruction `type(mavariable)` permet de connaître le type de `mavariable`.

Exemples :

```
>>> x = 9
>>> type(x)
<class 'int'>
>>> type(15)
<class 'int'>
>>> y = 11.25
>>> type(y)
<class 'float'>
>>> a = True
>>> type(a)
<class 'bool'>
>>> z = 2 + 3j
>>> type(z)
<class 'complex'>
>>> t = "bonjour"
>>> type(t)
<class 'str'>
```

¹¹ - « chaîne de caractères » est souvent raccourci en « chaîne ».

a) Expression littérale des données

La représentation littérale des données est la représentation sous forme de texte telle qu'on la trouve dans les programmes.

Pour les **nombres entiers** en base 10, on utilise simplement les chiffres décimaux et les signes + et - : **562**, **672**, **0**, **-75**, **+243**. On ne peut pas commencer un nombre à plusieurs chiffres par **0** car ceci est réservé à l'expression littérale d'entiers dans d'autres bases : en binaire (base 2, préfixe **0b**) **0b10011011**, en hexadécimal (base 16, préfixe **0x**) **0xf7e5a2b**, en octal (base 8, préfixe **0o**) **0o754**.

Pour les **nombres flottants** (ou nombres décimaux), toujours exprimés en base 10, on utilise un **.** pour séparer la partie entière de la partie décimale, et éventuellement une notation pour spécifier la puissance de 10 associée : **45.12e-6**, **-56E8**, **0.0**, **.0**, **-11.26562e-2**.

Pour les **chaînes de caractères**, on les encadre par des simples ou doubles guillemets : **"Un texte"**, **'Un autre texte'**. On peut y insérer des séquences d'échappement¹² introduites par un caractère **** suivi par un autre caractère définissant la séquence, **\t** pour une tabulation, **\n** pour un saut de ligne, **** pour un ****, **\"** pour un guillemet double (utile dans une chaîne encadrée par des guillemets doubles), **\'** pour un guillemet simple (utile dans une chaîne encadrée par des guillemets simples). On peut aussi exprimer des littéraux chaînes de caractères sur plusieurs lignes en utilisant au début et à la fin trois guillemets doubles ou trois guillemets simples :

```
"""Une chaîne sur plusieurs lignes
avec "des" retours et
des guillemets."""
```

Lorsque plusieurs littéraux chaîne de caractères se suivent Python les regroupe automatiquement en une seule chaîne.

```
>>> "un" "deux" 'trois' """quatre"""
'undeuxtroisquatre'
```

4) Déclaration et initialisation d'une variable

Dans la plupart des langages de programmation, pour pouvoir utiliser une variable, il faut procéder en deux temps :

- 1) **déclarer** la variable avec un certain **type**,
- 2) lui **affecter** une **valeur** (et éventuellement lui réaffecter une ou plusieurs nouvelles valeurs par la suite).

Par exemple, dans le langage C++, on doit écrire :

```
int x ;    // déclaration de x comme un nombre entier
x = 4 ;    // affectation de la valeur 4 à la variable x, grâce à l'opérateur d'affectation
x = 10 ;   // réaffectation d'une nouvelle valeur (ici, 10) à cette même variable x.
```

On ne peut alors affecter que des entiers à la variable **x**. Une instruction comme :

```
x = 12.5 ;
```

provoque une erreur.

En revanche, **en Python**, il n'est *pas nécessaire de déclarer explicitement les variables en précisant leur type*. Lors de l'opération d'affectation, le langage associe à la variable le type de la donnée qu'elle référence.

Exemple en python :

```
>>> x = 4          # déclaration de la variable x et affectation de la valeur 4 à cette
variable. Python lui donne automatiquement le type entier (int), puisque 4 est un entier
>>> type(x)
<class 'int'>
>>> x = 10         # réaffectation d'une nouvelle valeur à la variable x.
>>> type(x)
<class 'int'>
>>> x = 12.5       # réaffectation : python change automatiquement le type de la variable
x : elle devient de type flottant (float).
>>> type(x)
<class 'float'>
```

12 - Ces séquences d'échappement permettent aussi d'introduire des caractères par leur nom ou leur code Unicode, Par exemple, pour π : **"\N{GREEK SMALL LETTER PI}"**, **"\u03C0"**.

5) Le transtypage ou conversion de type

On peut changer le type d'une donnée en l'indiquant entre parenthèses, précédée du nom du nouveau type souhaité.

```
>>> float(3)
3.0
>>> str(3)
'3'
>>> int(3.7)
3
>>> str(3.4)
'3.4'
>>> float("3.4")
3.4
>>> bool(0)
False
>>> int(True)
1
```

Ce changement de type est appelé **transtypage** ou conversion de type, ou **cast** en anglais.

Certains transtypages ne sont pas autorisés (la valeur affectée à la variable ne peut pas être convertie vers le type désiré) :

```
>>> int("bonjour")
ValueError: invalid literal for int() with base 10: 'bonjour'
```

6) Les opérateurs

Afin de pouvoir effectuer des opérations sur des données, on utilise des opérateurs :

`+, -, *, /, //, %, **, <, >, ==, <=, >=, !=, and, or, not` (... etc)



L'action d'un opérateur dépend du type des données utilisées.

Voici quelques opérateurs python qui sont destinés à des données nombres entiers ou nombres flottants :

- `+` addition de nombres
- `-` soustraction de nombres
- `*` multiplication
- `**` élévation à la puissance
- `/` division décimale
- `//` quotient d'une division entière
- `%` reste d'une division entière

Et pour les chaînes de caractères :

- `+` concaténation de deux chaînes de caractères
- `*` répétition d'une chaîne de caractères (chaîne * entier)

Exemples : utilisons ces opérateurs dans l'interpréteur Shell Python :

```
>>> 3 + 7
10
>>> 3.2 + 7
10.2
>>> 3.2 - 7
-3.8
>>> 3*7
21
>>> 3/4
0.75
>>> 1/3
0.3333333333333333
>>> 2**4
16
```

```

>>> 9/2
4.5
>>> 9//2
4
>>> 9%2
1
>>> "mot1" + "mot2"
'mot1mot2'
>>> "mot1" + " " + "mot2"
'mot1 mot2'
>>> "bonjour ! " *4
" bonjour ! bonjour ! bonjour ! bonjour ! "
>>> "ha" * 5
'hahahahaha'

```

On peut affecter le résultat d'une opération (on parle d'expression) à une variable.

Exemple :

```

>>> y = 1 - 6
>>> y
-5
>>> z = y * 5
>>> z
-25
>>> t = y / z
>>> t
0.2

```

On peut aussi réaffecter à une variable, le résultat d'une opération dans lequel cette variable elle-même intervient.

Exemple : réaffectation à une variable x, de son ancienne valeur, à laquelle on ajoute 100 :

```

>>> x = 4
>>> x
4
>>> x = x + 100
>>> x
104

```

L'affectation se fait en deux temps :

- 1) **évaluation** (c'est à dire calcul) de l'**expression** située à *droite de l'opérateur d'affectation* (selon des règles de priorité de calcul correspondant au type des données concernées). Le résultat de cette évaluation est la donnée (le contenu) qui va être « rangée » dans la variable.
- 2) **rangement** du résultat de cette évaluation dans la **variable** (association du nom à la donnée), qui est obligatoirement la partie située à *gauche de l'opérateur d'affectation*.

Rappelons qu'en algorithmique, l'opérateur d'affectation est représenté par une flèche gauche ($x \leftarrow 4$), ce qui illustre bien le fait que c'est la partie située à *droite* de l'opérateur d'affectation qui va être stockée dans la partie située à *gauche* de l'opérateur d'affectation.

Vocabulaire :

Incrémenter une variable signifie lui ajouter une valeur numérique (en général, la valeur 1).

```

x = x + 1    # on incrémente x
x += 1       # l'opérateur += permet de faire la même chose

```

Décrémenter une variable signifie lui ôter une valeur numérique (en général, la valeur 1).

```

x = x - 1    # on décrémente x
x -= 1       # l'opérateur -= permet de faire la même chose

```

III - Les programmes python - les modules

Un *fichier script python*, d'extension **.py**, se nomme aussi un **module**¹³.

Un module python fait généralement appel à des éléments qui sont présents dans d'autres modules python ; *un programme python peut donc se composer d'un ou de plusieurs modules*.

Nos programmes utiliseront généralement des modules déjà écrits par d'autres auteurs, comme les modules `math`, `cmath`, `random`, `turtle`, `matplotlib`, `numpy`, etc (soit ces modules font déjà partie du langage python¹⁴, soit on peut les trouver sur internet).

1) Exemples

a) Exemple 1 - calculs

On va calculer et afficher à l'écran, l'aire d'un disque et le volume d'une sphère ayant pour rayon un entier choisi aléatoirement.

```
1. import random          # on veut utiliser le module random
2. from math import pi    # on veut utiliser la valeur du nombre pi qui est enregistrée
   dans le module math
3. rayon = random.randrange(1,100)    # choix d'un entier au hasard entre 1 et 99
4. aire_disque = pi*rayon**2    # calcul de l'aire d'un disque et affectation du résultat à
   la variable aire_disque
5. vol_sphere = 4/3*pi*rayon**3    # calcul du volume d'une sphère et affectation du
   résultat à la variable vol_sphere
6. # on affiche les résultats à l'écran
7. print("La surface d'un disque de rayon", rayon, "m est de", aire_disque,"m2")
8. print("Le volume d'une sphère de rayon", rayon, "m est de", vol_sphere,"m3")
```

b) Exemple 2 - graphisme

On va dessiner à l'écran un cercle dont le rayon sera un nombre flottant choisi au hasard, puis, on va le colorier en vert.

```
1. from random import randrange    # on veut utiliser le module random
2. import turtle                  # on veut utiliser le module turtle ("tortue")
3. monrayon = randrange(1,100)    # choix d'un entier au hasard entre 1 et 99
4. print("le rayon choisi au hasard est", monrayon) #on affiche la valeur du rayon choisi
5. turtle.color('black','green') #le curseur écrira en noir et le coloriage sera vert
6. turtle.begin_fill()           #on indique que la forme choisie devra être remplie
7. turtle.circle(monrayon)       #on trace un cercle ayant comme rayon, le nombre flottant choisi
   au hasard précédemment
8. turtle.end_fill()             #on termine le coloriage
```

2) Suivre l'exécution d'un programme : le tableau de suivi

Lorsque les instructions contiennent des variables, il est conseillé de suivre (ou "tracer") la valeur de ces variables au fur et à mesure des instructions, dans un **tableau de suivi**.

Un tableau de suivi doit contenir une colonne par variable. On fait figurer les affichages soit en fin de tableau (s'il n'y a qu'un affichage final) soit dans une colonne du tableau (lorsqu'il y a plusieurs affichages tout au long du programme). Pour une meilleure lisibilité, on peut éventuellement indiquer les numéros de ligne du programme dans une première colonne.

13 - Il est aussi possible d'écrire des modules Python en langage C, afin de bénéficier de la vitesse d'exécution des programmes traduits en langage machine. C'est ainsi qu'est écrit une partie des bibliothèques standard Python, ainsi que des bibliothèques de calcul comme *numpy*.

14 - On dit que Python est « Batteries Included » car il est installé en standard avec de nombreux modules.

Exemple de tableau de suivi pour le programme Exemple 1 - calculs :

n° ligne	rayon	aire_disque	vol_sphere	Affichages
1				
2				
3	42			
4		5541.769440 932 395		
5			310 339.0886 922 141	
6				
7				La surface d'un disque de rayon 42 m est de 5541.769440 932 395 m2
8				Le volume d'une sphère de rayon 42 m est de 310 339.0886 922 141 m3

3) Tester un programme : le jeu d'essais

Un jeu d'essais est un ensemble de valeurs sur lesquelles on prévoit de tester le programme pour vérifier qu'il fonctionne dans tous les cas. On doit tester tous les cas d'utilisation du programme, sans oublier les cas limite. On doit préciser quels résultats sont attendus pour chaque essai.

Exercice : reprenons l'algorithme précédent qui permettait de déterminer le résultat d'un semestre (cf pages 5 et 11). Pour vérifier qu'il fonctionne correctement, on le testera avec le jeu d'essais suivant (à remplir) :

Essai n°	Description de l'essai	Résultat attendu

4) Présenter un module : en-tête, commentaires, résultats d'exécution

Au début du module doit figurer un en-tête de module qui contient le nom du fichier, le nom des auteurs, la date ainsi qu'une documentation c'est à dire une explication sur ce que fait le script :

```

1. #!/usr/bin/python3
2. # -*- coding: UTF-8 -*-
3. """
4. Documentation du module(appelée aussi DocString)
5. Il s'agit d'expliquer ce que le module contient et à quoi il sert.
6. Il peut être affiché avec help(nommodule) ou help("nommodule"),
7. ou bien utilisé par un outil de génération de documentation
8. externe (produisant du html, du pdf...).
9. """
10. # fichier: nom_du_fichier.py (indiquer le nom du fichier)
11. # auteurs : (indiquer les noms des auteurs du module)
12. # date : (indiquer la date)

```

Ensuite vient le programme proprement dit, c'est à dire les instructions de l'algorithme traduites en langage python :

```

13. # programme principal
14.
15. import random          # on veut utiliser le module random
16. from math import pi    # on veut utiliser la valeur du nombre pi qui est enregistrée
    dans le module math
17. rayon = random.randrange(1,100)    # choix d'un entier au hasard entre 1 et 99
18. aire_disque = pi*rayon**2    # calcul de l'aire d'un disque et affectation du résultat à
    la variable aire_disque
19. vol_sphere = 4/3*pi*rayon**3    # calcul du volume d'une sphère et affectation du
    résultat à la variable vol_sphere
20. # on affiche les résultats à l'écran
21. print("La surface d'un disque de rayon", rayon, "m est de", aire_disque,"m2")

```

```
22. print("Le volume d'une sphère de rayon", rayon, "m est de", vol_sphere,"m3")
```

À la fin du module, après avoir enregistré puis exécuté le programme, copiez-collez les résultats de l'exécution⁴ entre des triple-guillemets, comme dans l'exemple ci-dessous.

```
23. """
24. ---- Résultats de l'exécution ----
25. >>>
26. La surface d'un disque de rayon 63 m est de 12468.981242097889 m2
27. Le volume d'une sphère de rayon 63 m est de 1047394.4243362226 m3
28. >>> ===== RESTART =====
29. >>>
30. La surface d'un disque de rayon 17 m est de 907.9202768874502 m2
31. Le volume d'une sphère de rayon 17 m est de 20579.526276115535 m3
32. >>> ===== RESTART =====
33. >>>
34. La surface d'un disque de rayon 6 m est de 113.09733552923255 m2
35. Le volume d'une sphère de rayon 6 m est de 904.7786842338603 m3
36. """
```

Des commentaires doivent figurer dans les programmes pour en faciliter la relecture : pour les autres programmeurs afin de comprendre ce que vous avez voulu faire... mais aussi pour vous lorsque vous aurez à reprendre vos propres programmes plus tard.

Les commentaires ne sont pas interprétés comme des instructions du langage, ils sont simplement ignorés par Python et n'ont pas d'impact sur la vitesse d'exécution. Ils sont introduits par un caractère # (tout ce qui suit le # sur la même ligne est un commentaire). Vous pouvez aussi les écrire sous la forme d'une chaîne de caractère entre triple-guillemets (""" suivis de """) ce qui permet de les écrire sur plusieurs lignes.

Tous vos programmes devront être suffisamment commentés (mais pas trop : il ne faut pas abuser des commentaires sous peine de perdre du temps et de surcharger inutilement le code source du programme).

Un exemple de commentaire inutile :

```
x = 1 # Stocke 1 dans x
```

Et un exemple de commentaire utile :

```
x = (pt1.x + pt2.x) / 2 # Abscisse milieu des deux points.
```


IV - Les fonctions : utilisation

En algorithmique, une **procédure** est un regroupement d'instructions que l'on nomme par un nom afin de pouvoir les exécuter de manière plus simple et/ou répétitive. Une **fonction** est une procédure qui calcule et qui retourne un *résultat*.

Exemples :

- `print()` est une *procédure* qui permet de faire des affichages à l'écran ; `turtle.circle()` est une *procédure* qui permet de tracer un cercle à l'écran.
- `math.sin()` et `math.cos()` sont des *fonctions* qui permettent de calculer le sinus et le cosinus d'un nombre et retournent leur résultat sous la forme d'un nombre flottant ;
- `float()`, `int()`, `str()`, `bool()` sont des *fonctions* qui permettent de convertir une donnée d'un type dans un autre type, et retournent leur résultat sous la forme de la donnée dans le type désiré ;
- `random.randrange()` est une *fonction* qui permet de choisir un entier au hasard.

En python, nous dirons indifféremment « **fonction** » pour parler de **fonctions** ou de **procédures** (en Python une procédure est une fonction qui retourne implicitement la valeur **None**).

Une fonction est parfois appelée une **méthode** (nous verrons ultérieurement dans quels cas).

1) Importation de fonctions prédéfinies depuis des "bibliothèques"

Certains modules contiennent des fonctions prédéfinies destinées à être utilisées dans d'autres programmes ou modules python. On appelle ces modules des **bibliothèques** (ou librairies en anglais).

Par exemple, on trouve dans le module `math`, les fonctions prédéfinies `sqrt`, `cos`, `sin`, etc. Pour pouvoir les appeler et les utiliser, il faut les importer. On écrira :

```
import nomModule
```

ou bien

```
from nomModule import nomFonction1, nomFonction2, nomFonction3
```

Un exemple dans un Shell Python :

```
>>> import math
>>> math.sin(0)
0.0
>>> math.sqrt(2)      # on calcule la racine carrée (square root) de 2
1.4142135623730951
>>> from math import sin
>>> sin(0)
0.0
>>> from math import sqrt
>>> sqrt(2)
1.4142135623730951
>>> from math import pi, cos
>>> pi
3.141592653589793
>>> cos(pi)
-1.0
```

Le module `cmath` contient des fonctions destinées à manipuler les nombres complexes.

```
>>> import cmath
>>> cmath.sqrt(-2)
1.4142135623730951j
```

On peut importer tout le contenu d'un module à la fois en utilisant le caractère `*`. Mais *cela est fortement déconseillé*, car on ne pourrait alors plus faire la distinction entre des fonctions ou variables provenant de différents modules, s'ils sont nommés de façon identique.

Par exemple si nous écrivons ce qui suit, nous ne pourrons plus différencier la fonction `sqrt()` du module `math` de la fonction `sqrt()` du module `cmath` :

```
>>> from math import *
>>> from cmath import *
```

2) Fonctions d'aide

Les fonctions `help()` et `dir()` permettent d'obtenir de l'aide sur des fonctions prédéfinies ou sur des modules tout entiers. La fonction `help()` affiche le contenu des chaînes de documentation DocString (voir page 20). La fonction `dir()` affiche la liste de toutes les fonctions et variables disponibles dans un module. Vous pouvez essayer :

```
>>> import math
>>> dir(math)

>>> help(math)

>>> help(math.sin)

>>> help(math.sqrt)

>>> import random
>>> dir(random)

>>> help(random.randrange)

>>> help(random.uniform)
```

Par exemple pour écrire le programme d'exemple 2 ci-dessus (cf. page 19) on a affiché de l'aide sur certaines fonctions du module `turtle`, de la façon suivante :

```
>>> dir(turtle)
>>> help(turtle.circle)
>>> help(turtle.begin_fill)
>>> help(turtle.end_fill)
>>> help(turtle.color)
```

3) Fonctions d'entrée et sortie

Afin que l'utilisateur puisse entrer des données dans les variables du programme et visualiser les résultats calculés par le programme, le programmeur utilise des fonctions d'entrée et de sortie pour écrire les instructions du programme.

La fonction de sortie `print()` permet d'afficher à l'écran (à l'attention de l'utilisateur) le contenu des variables, le résultat des calculs, etc. C'est en réalité une procédure car elle ne retourne pas de résultat.

La fonction d'entrée `input()` permet de récupérer et d'affecter aux variables la saisie de l'utilisateur au clavier. En python3, la donnée récupérée par cette fonction `input` est de type chaîne de caractères. Si on souhaite obtenir un nombre (et non pas une chaîne de caractères), il faudra *convertir cette donnée* en nombre entier ou flottant.

Les fonctions `print()` et `input()` sont surtout utiles dans les programmes (fichiers `.py`) mais ne sont pas très utiles dans l'interpréteur Shell Python (dans celui on peut directement affecter une valeur, du type désiré, à une variable ; et il affiche automatiquement le résultat d'une expression saisie, par exemple une simple variable).

Exemple d'utilisation dans un script :

```
nb = input("entrez votre nombre")
nb = float(nb)
var = sin(nb)
print("le sinus de votre nombre est", var)
```

D'autres fonctions d'entrée et de sortie permettent d'écrire et de lire directement dans des fichiers stockés sur le disque dur (nous les étudierons plus tard dans le semestre). Il s'agit de `f.read()`, `f.readline()`, `readlines()`, `f.write()`, `f.writelines()`.

4) Appeler une fonction prédéfinie

L'**appel d'une fonction** consiste à écrire son nom dans l'interpréteur ou dans l'éditeur, suivi de parenthèses entre lesquelles on indique des *données* ou *variables* qui sont les **arguments**¹⁵ de la fonction.

On dit qu'on **pass**e des arguments à la fonction, et que la fonction **reçoit** ou **prend** des arguments.

Une fonction peut recevoir entre zéro et plusieurs arguments :

```
nomDeLaFonction(argument1, argument2, argument3)
```

Exemples :

```
print("coucou")
print("il y a environ", 170, "étudiants en s1")
float("3.4")
```

Si la fonction appartient à un **module** particulier, il faut *importer ce module*, et préfixer le nom de la fonction par le nom du module :

```
import nomModule
nomModule.nomDeLaFonction(argument1, argument2, argument3)
```

Exemples :

```
import turtle
turtle.circle(100)
import math
math.sin(0)
math.cos(math.pi)
```

Les **arguments** fournis aux fonctions lors des appels peuvent être des variables :

```
>>> nb = 0
>>> sin(nb)
0.0
>>> nb = pi
>>> cos(nb)
-1.0
>>> chaine_de_caracteres = "coucou"
>>> print(chaine_de_caracteres)
coucou
>>> nb = float(input("entrez votre nombre "))
entrez votre nombre ... (nous devons ici saisir un nombre au clavier, par exemple 2)
>>> sin(nb)
0.9092974268256817 (le sinus du nombre précédemment saisi s'affiche. Dans notre exemple, c'est
sin(2) qui a été calculé puisque nb a reçu la valeur 2 à l'étape précédente)
>>> longueur = 150
>>> turtle.circle(longueur) (un cercle de rayon 150 pixel est tracé)
```

Certaines fonctions prennent un *nombre d'arguments bien précis*. Par exemple, la fonction `cos()` et la fonction `sin()` doivent recevoir exactement un seul argument.

D'autres fonctions peuvent recevoir un *nombre variable d'arguments* : par exemple, la fonction `print()` peut recevoir un nombre quelconque d'arguments (aucun, un ou plusieurs arguments). On dit que les arguments de telles fonctions sont *optionnels*.

5) Valeur retournée par une fonction

Toute fonction retourne une valeur. Cette valeur retournée est le résultat du "travail" de la fonction (la fonction détermine ou calcule cette valeur).

On obtient cette valeur retournée en appelant la fonction. La valeur retournée peut être affectée à une variable, utilisée directement dans un calcul, ou rester inutilisée si le programme n'en a pas besoin.

15 - Les arguments sont associés aux **paramètres** listés dans la définition de la fonction, il arrive souvent qu'on utilise abusivement le terme « paramètres » au lieu du terme « arguments ».

Exemples :

```
>>> from math import sin, cos, pi
>>> var = sin(0)
>>> var
0.0
>>> sin(pi)+cos(pi)
-0.9999999999999999
>>> var = int("33")
>>> var
33
>>> var = str(45)
>>> var
'45'
>>> input("appuyez sur une touche pour continuer...")
appuyez sur une touche pour continuer...
```

Les *procédures* sont des fonctions dont le travail ne consiste pas en un calcul d'une donnée. Mais même si elles ne calculent rien, les procédures en python retournent une valeur qui est la valeur **None** (c'est pourquoi on peut se permettre de les appeler, elles aussi, des « fonctions »).

Exemple :

```
>>> var = print("coucou")      (on stocke la valeur de retour de l'appel à print dans var)
coucou
>>> var
None                           (la procédure print( ) retourne bien None)
>>> type(var)
<class 'NoneType'>
```

V - Les booléens et l'instruction conditionnelle if

1) Le type booléen

Le type *booléen* ou *bool* est un type de données. Un booléen ne peut prendre que 2 valeurs : **vrai** (True) ou **faux** (False).

Les majuscules sont importantes : `true` et `false` ne sont pas reconnues par le langage, il faut écrire `True` et `False`.

Dans certaines représentations (comme par exemple dans l'algèbre de Boole), le booléen faux est représenté par un 0 tandis que le booléen vrai est représenté par un 1.

a) Opérateurs de comparaison

En python, les **opérateurs de comparaison** s'appliquent à des données (numériques, chaînes de caractères...) et produisent un **résultat booléen**. Ils permettent de faire des *tests*. Le résultat de l'opération est `True` si le test est vrai, `False` si le test est faux.

<code>==</code>	signifie "est égal à"
<code>!=</code>	signifie "est différent de"
<code>< , <= , > , >=</code>	signifient respectivement "est strictement inférieur, inférieur ou égal, strictement supérieur, supérieur ou égal".

Exemples :

```
>>> 4 == 4
True
>>> 4 == 1
False
>>> 4 != 1
True
>>> 4*3 == 2*6
True
```

Attention : ne pas confondre l'opérateur de test d'égalité `==` avec l'opérateur d'affectation `=`

Exemple :

```
>>> 4 == 4
True
>>> 4 = 4
SyntaxError: can't assign to literal
```

Remarque : comme python fait du calcul numérique approché (et pas du calcul formel), on peut obtenir des résultats erronés :

```
>>> from math import sin
>>> from math import pi
>>> sin(pi)
1.2246467991473532e-16          (1,2246...×10-16)
>>> sin(pi) == 0
False
```

b) Opérateurs sur les booléens

Les opérateurs python qui s'appliquent aux booléens s'appellent **opérateurs logiques**. Ce sont :

not (NON), **and** (ET) et **or** (OU).

<code>not</code>	signifie "NON"
<code>and</code>	signifie "ET"
<code>or</code>	signifie "OU"

Ils opèrent de la façon suivante : soient A et B des propositions (expressions pouvant prendre la valeur vrai ou faux) :

- “ non A ” est vrai si et seulement si A est faux
- “ non A ” est faux si et seulement si A est vrai
- “ A et B ” est vrai si et seulement si (A est vraie) et (B est vraie)
- “ A et B ” est faux si et seulement si (A est fausse) ou (B est fausse)
- “ A ou B ” est vrai si et seulement si (A est vraie) ou (B est vraie)
- “ A ou B ” est faux si et seulement si (A est faux) et (B est faux)

Exemples :

```
>>> 4 == 4 and 2 != 15
True
>>> 4 == 4 and 2 == 15
False
>>> 4 == 4 or 2 == 15
True
>>> 4 == -1 or 2 == 15
False
>>> not(3 < 15 )
False
>>> not(1<2 and 3<5)
False
```

c) Tables de vérité

c.1) Opérateur NON (not en python)

a	not a
False (Faux)	True (Vrai)
True (Vrai)	False (Faux)

c.2) Opérateur ET (and en python)

a	b	a and b
False	False	False (Faux)
False	True	False (Faux)
True	False	False (Faux)
True	True	True (Vrai)

c.3) Opérateur OU (or en python)

a	b	a or b
False	False	False (Faux)
False	True	True (Vrai)
True	False	True (Vrai)
True	True	True (Vrai)

2) Algèbre de Boole et logique booléenne

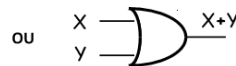
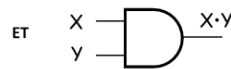
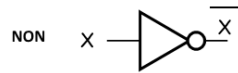
En algèbre de Boole, on peut représenter le Vrai et le Faux par des éléments de l'ensemble {0 ; 1}

- 0 représente le Faux
- 1 représente le Vrai

Ainsi, on peut représenter des variables d'entrée X et Y par des éléments de {0 ; 1} et les opérateurs logiques NON, ET, OU peuvent être considérés comme des fonctions de une ou plusieurs variables de {0 ; 1}, à valeurs dans l'ensemble {0 ; 1}.

a) Opérateurs NON, ET, OU

En informatique d'instrumentation (voir semestre 2) ces opérateurs sont aussi appelés « portes logiques » et on les représente dans les schémas de la façon suivante :



b) Propriétés

soient A, B et C des booléens.

b.1) Propriétés de NON

- **non (non (A)) = A** *involution*

b.2) Propriétés de ET

- **A et B = B et A** *commutativité*
- **A et (B et C) = (A et B) et C** *associativité*
- **A et non(A) = Faux**
- **A et Faux = Faux**
- **A et Vrai = A**
- **A et A = A** *idempotence*

b.3) Propriétés de OU

- **A ou B = B ou A** *commutativité*
- **A ou (B ou C) = (A ou B) ou C** *associativité*
- **A ou non(A) = Vrai** *tautologie*
- **A ou Faux = A**
- **A ou Vrai = Vrai**
- **A ou A = A** *idempotence*

b.4) Distributivité

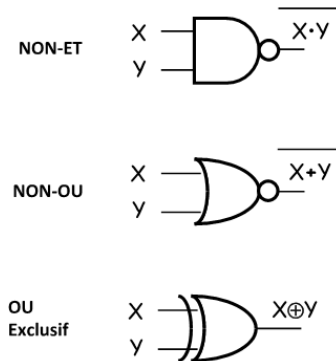
- **A et (B ou C) = (A et B) ou (A et C)** *distributivité du ET par rapport au OU*
- **A ou (B et C) = (A ou B) et (A ou C)** *distributivité du OU par rapport au ET*
- **A ou (A et B) = A** *absorption*

b.5) Lois de De Morgan

- **non(A ou B) = non(A) et non(B)**
- **non(A et B) = non(A) ou non(B)**

c) Autres portes logiques

Vous utiliserez d'autres portes logiques en informatique d'instrumentation, au semestre 2 :



- X OU-Exclusif Y est vrai si **une et une seule** des variables X, Y est vraie (aussi noté XOR) – c'est le OU du restaurant dans « fromage ou dessert ».
- X NON-ET Y est vrai si non(X et Y) est vrai (aussi noté NAND).
- X NON-OU Y est vrai si non(X ou Y) est vrai (aussi noté NOR). Cet opérateur NON-OU est aussi noté NOR.

Voir le tableau récapitulatif dans l'annexe Récapitulatif des opérateurs booléens en page 73.

3) L'instruction conditionnelle if

L'instruction **if** est une instruction composée. Elle signifie « si ».

Sa syntaxe est la suivante :

```
if condition1 :
    bloc1
elif condition2 :
    bloc2
elif condition3 :
    bloc3
else :
    bloc4
```

Les conditions sont des valeurs logiques booléennes (True ou False). Si la condition est une valeur d'un autre type, alors celle-ci est convertie automatiquement en booléen.

Lors de la conversion automatique d'une valeur vers un booléen : toutes les valeurs qui sont nulles (0, 0.0, 0+0j), ou vides (" ", ' ', [], {}), ou bien **None** sont considérées comme False. Toutes les autres valeurs sont considérées comme True.

Seule la ligne **if** condition1 est obligatoire. On peut mettre autant de lignes **elif** condition_n que l'on souhaite (entre 0 et plusieurs). On peut mettre au maximum (aucune ou une) une ligne **else**. On ne doit pas préciser de condition après le mot clé **else**.

Exemple :

```
if note >= 16 :
    mention = "TB"
elif note >= 14 :
    mention = "B"
elif note >= 12 :
    mention = "AB"
elif note >= 10 :
    mention = "passable"
else :
    mention = "ajourné"
print("Voici votre résultat :", mention)
```


VI - Les séquences - l'instruction de boucle for

1) Boucle for

L'instruction composée **for** permet de répéter un bloc d'instructions un certain nombre de fois.

Exemple 1 :

```
print("voici la table de 8")
for k in range(10):
    print("8 fois ", k, " = ", 8*k)
```

Affichage lors de l'exécution :

```
voici la table de 8
8 fois 0 = 0
8 fois 1 = 8
8 fois 2 = 16
8 fois 3 = 24
8 fois 4 = 32
8 fois 5 = 40
8 fois 6 = 48
8 fois 7 = 56
8 fois 8 = 64
8 fois 9 = 72
```

Exemple 2 :

```
mot = "informatique"
for lettre in mot :
    print(lettre)
```

Affichage lors de l'exécution :

```
i
n
f
o
r
m
a
t
i
q
u
e
```

a) Syntaxe

```
for variable_iterateur in séquence :
    bloc d'instructions secondaires...
```

Les instructions du bloc secondaire (aussi appelé corps de la boucle) sont répétées (itérées) autant de fois que la *séquence* contient d'éléments. À chaque passage la *variable itérateur* prend la valeur suivante dans la *séquence* — l'affectation de la valeur de la *séquence* à la *variable itérateur* est réalisée automatiquement.

👉 Attention : il est **INTERDIT** de modifier la variable itérateur dans le bloc secondaire (aussi appelé corps) d'une boucle for.

```
for k in range(3)
    print(k)
    k = k+1    # A NE PAS FAIRE même si le langage python semble l'accepter
```

2) Séquences ou types itérables

Un type "itérable" est un type de donnée sur lequel on peut faire une itération, c'est à dire que l'on peut énumérer les données qu'il contient les unes après les autres, en séquence. Tous les types itérables peuvent être parcourus avec une boucle **for**.

a) Générateur range

Soient n et p et q des entiers relatifs (c'est à dire positifs ou négatifs).
`range(n)` ► génère la suite des entiers de 0 inclus à n exclu.
`range(p,n)` ► génère la suite des entiers de p inclus à n exclu.

Exemples :

```
for i in range(10):
    print(i)
```

Affichage à l'exécution :

```
0
1
2
3
4
5
6
7
8
9
```

Exemple :

```
for i in range(3,8):
    print(i, end = ",")
```

Affichage à l'exécution :

```
3,4,5,6,7,
```

`range(-2,2)` génère la suite -2,-1,0,1

`range(4,2)` génère une suite vide

`range(p,n,q)` ► génère la suite des entiers de p inclus à n exclu, avec un pas égal à q
 ◦ si $q > 0$: la suite est construite en ordre croissant
 ◦ si $q < 0$: la suite est construite en ordre décroissant

Exemples :

- `range(2,12,2)` génère la suite 2,4,6,8,10
- `range(0,12,3)` génère la suite 0,3,6,9
- `range(15,10,-1)` génère la suite 15,14,13,12,11

b) Les listes

En python, le type `list` est un type de données qui permet de former une **suite ordonnée d'éléments**. Les éléments d'une même liste peuvent être des données de tous types¹⁶.

On écrit les éléments d'une liste python entre deux crochets, séparés par des virgules :

```
[element1, element2, element3, element4]
```

Exemple :

```
>>> maliste = ['a','b',1,2,3,"toto",9.5]
>>> type(maliste)
<class 'list'>
```

b.1) Opérations et syntaxes de base

- `[]` est la liste vide

16 - Dans d'autres langages, comme le C/C++ ou le Pascal, les listes, appelées **tableaux**, ne peuvent contenir qu'un seul type de données à la fois (on peut avoir un tableau d'entiers, un tableau de flottants, un tableau de chaînes de caractères, un tableau de tableaux d'entiers, etc).

- Le **nombre d'éléments** d'une liste s'appelle sa **longueur**. Elle est renvoyée par la fonction `len()`¹⁷ appliquée à la liste :

```
>>> len(maliste)
7
```

- Les éléments d'une liste sont repérés par leur **indice**. Les indices d'une liste commencent à 0 et se terminent à (longueur-1), Python détecte automatiquement l'utilisation d'index invalides et génère une erreur (exception).

`maliste[k]` est l'élément d'indice `k` dans `maliste`.

```
>>> maliste[0]
'a'
>>> maliste[1]
'b'
>>> maliste[6]
9.5
>>> maliste[7]
IndexError: list index out of range
```

- On peut tester l'**appartenance** d'un élément à une liste grâce à l'**opérateur** `in` :

élément `in` `maliste` renvoie le booléen `True` si l'élément est dans la liste, `False` sinon

```
>>> "toto" in maliste
True
>>> "robert" in maliste
False
```

- On peut **modifier les éléments** d'une liste un par un en leur affectant une nouvelle valeur.

```
>>> maliste[1] = 15
>>> maliste
['a', 15, 1, 2, 3, "toto", 9.5]
```

- On peut *créer une liste en une seule instruction* de la façon suivante (on parle de *liste en compréhension*) :

```
>>> listeCarres = [ k**2 for k in range(100) ]
```

- Il y a deux méthodes pour itérer sur une liste avec une boucle `for` :

- méthode 1 : **itérer sur les éléments** de la liste (cette façon de faire est propre au langage python).

```
>>> for element in maliste :
    print(element)
    (appuyer une 2e fois sur entrée pour terminer la saisie du bloc d'instructions secondaire)
```

```
a
15
1
2
3
toto
9.5
```

- méthode 2 : **itérer sur les indices des éléments** de la liste, c'est à dire sur une suite d'entiers (cette façon de faire est commune à la plupart des langages de programmation). On accède alors indirectement aux éléments en utilisant les indices, cela permet d'accéder dans le corps de la boucle à l'élément précédent ou à l'élément suivant, ou encore d'aller changer dans la liste la valeur de l'élément .

```
>>> for k in range(len(maliste)):
    print(k, maliste[k])
    (appuyer une 2e fois sur entrée pour terminer la saisie du bloc d'instructions secondaire)
```

```
0 a
1 15
2 1
3 2
```

17 - En python, la longueur de la liste est gérée automatiquement par le langage (le programmeur n'a pas à s'en occuper), contrairement aux tableaux C/C++ où la longueur de la liste doit être gérée par le programmeur. En C++ les types de la Standard Templates Library offrent des services de plus haut niveau que les tableaux de base, leur utilisation est fortement conseillée - cf. STL Containers.

```
4 3
5 toto
6 9.5
```

- On peut **ajouter un élément à la fin** d'une liste grâce à la méthode `append()` (ce genre d'instruction, appelée *méthode*, correspond à une fonction spécifique de la liste que l'on appelle avec la syntaxe `variable.méthode()`).

```
>>> maliste.append("fleur")
>>> maliste
['a',15,1,2,3,"toto",9.5,"fleur"]
>>> len(maliste)
8
```

- On peut **supprimer un élément** grâce aux méthodes `pop()` ou `remove()`.

```
>>> maliste.pop()
'fleur'
>>> maliste
['a',15,1,2,3,"toto",9.5]
>>> maliste.pop(0)
'a'
>>> maliste
[15, 1, 2, 3, 'toto', 9.5]
>>> maliste.pop(4)
'toto'
>>> maliste
[15, 1, 2, 3, 9.5]
>>> maliste.remove(15)
>>> maliste.remove(9.5)
>>> maliste
[1,2,3]
```

- L'opérateur `+` permet de **concaténer des listes**.

L'opérateur `*` permet de « multiplier » une liste par un entier, ce qui a pour effet de **dupliquer le contenu** de cette liste.

```
>>> liste2 = [10,'a']
>>> maliste + liste2
[1, 2, 3, 10, 'a']
>>> liste2 * 3
[10, 'a', 10, 'a', 10, 'a']
```

- Une liste peut elle-même **contenir une ou plusieurs listes** :

```
>>> liste4 = ['a',1,[10,20,30],50,'c',['z','x']]
>>> liste4[2]
[10, 20, 30]
>>> liste4[2][0]
10
>>> liste4[2][1]
20
>>> liste4[5][1]
'x'
```

Ceci permet de définir des **tableaux à 2 dimensions** (ou plus) : un tableau à p lignes et n colonnes se représente par une liste contenant p sous-listes avec chacune n éléments (ou par une liste de n listes de p éléments chacune si on veut indexer d'abord sur la colonne). Par exemple, le tableau :

11	12	13
21	22	23
31	32	33
41	42	43

se représente par la liste : `[[11,12,13],[21,22,23],[31,32,33],[41,42,43]]`

- **Extraction d'une sous-liste** à partir d'une liste :

`maliste[p:n:q]` renvoie la sous-liste formée des éléments de `maliste` d'indice p inclus, jusqu'à l'indice n exclu, avec un pas égal à q .

```
>>> maliste = ['a','b',10,20,30,"toto",9.5]
```

```
>>> maliste[2:5]
[10, 20, 30]
>>> maliste[2:7:2]
[10, 30, 9.5]
>>> maliste[::-1]
[9.5, 'toto', 30, 20, 10, 'b', 'a']
>>> maliste[::-2]
[9.5, 30, 10, 'a']
```

- On obtient la **liste de toutes les méthodes** (ou fonctions) du type `list` en tapant `dir()`. L'instruction `help()` permet d'obtenir de l'aide sur une des méthodes. Ces deux outils sont utilisables quels que soient le type ou la donnée en Python.

```
dir(list)
help(list)
help(list.insert)
help(list.sort)
help(list.reverse)
```

Nous pouvons tester dans l'interpréteur les méthodes de liste `insert`, `sort`, `reverse` :

```
>>> listeexemple=['a','b',55,'c',15]
>>> listeexemple.reverse()
>>> listeexemple
[15, 'c', 55, 'b', 'a']
>>> listeexemple.insert(1,"toto")
>>> listeexemple
[15, 'toto', 'c', 55, 'b', 'a']
>>> listeexemple[2:2] = ["ici","zzz",88]
>>> listeexemple
[15, 'toto', 'ici', 'zzz', 88, 'c', 55, 'b', 'a']
>>> listeexemple.sort()
TypeError: unorderable types: str() < int()
```

Sur la dernière ligne, une erreur est signalée : la liste mélange des données qui ne sont pas comparables (ici chaînes et entiers), et donc ne peuvent être triées.

- On peut obtenir le **plus grand** élément ou le **plus petit** élément d'une liste (à condition que les éléments de celle-ci soient comparables) à l'aide des fonctions `max` et `min`. On peut faire la **somme des éléments** d'une liste (à condition qu'ils soient tous numériques) à l'aide de la fonction `sum`.

```
>>> autreliste = [15,18,5,-1,7]
>>> max(autreliste)
18
>>> help(max)
>>> autreliste.sort()
>>> autreliste
[-1,5,7,15,18]
>>> sum(autreliste)
44
>>> uneliste=["ara","zebu","aaa","zebre","souris"]
>>> uneliste.sort()
>>> uneliste
['aaa', 'ara', 'souris', 'zebre', 'zebu']
>>> max(uneliste)
'zebu'
```

- Il est possible de **convertir en liste**, des données de type `range`, chaîne de caractères, ou d'un autre type séquence :

```
>>> list(range(3))
[0, 1, 2]
>>> list("bonjour tout le monde")
['b', 'o', 'n', 'j', 'o', 'u', 'r', ' ', 't', 'o', 'u', 't', ' ', 'l', 'e', ' ', 'm', 'o', 'n', 'd', 'e']
```

b.2) Listes en compréhension

Il est très courant de *construire une liste de résultats* dans une boucle `for`. Par exemple pour construire la liste des carrés des nombres de 1 à 10 :

```
lst1 = []
for i in range(1,11):
    lst1.append(i ** 2)
```

Ou bien la même chose, en excluant les carrés de multiples de 3 :

```
lst2 = []
for i in range(1,11):
    if i % 3 != 0:
        lst2.append(i ** 2)
```

Ou encore convertir une liste de chaînes de caractères *représentant* des nombres entiers en nombres entiers (sur lesquels on peut faire des calculs numériques) :

```
lst3 = []
for v in ["23", "-12", "0", "167", "-175", "10562", "32", "438", "965", "42"]:
    lst3.append(int(v))
```

Python a repris la notion de *définition en compréhension* d'ensembles (par opposition à la *définition en extension* d'ensembles où on énumère chacun des éléments), en l'appliquant aux listes via l'inclusion dans la définition de liste d'instructions de boucle **for** et éventuellement d'instructions de test **if**. Les trois exemples ci-dessus peuvent s'écrire respectivement :

```
lst1 = [i**2 for i in range(1,11)]
lst2 = [i**2 for i in range(1,11) if i%3!= 0]
lst3 = [int(v) for v in ["23", "-12", "0", "167", "-175", "10562", "32", "438", "965", "42"]]
```

On peut utiliser plusieurs niveaux de **for** et plusieurs niveaux de **if** dans une telle construction :

```
>>> [(x,y) for x in range(1,5) for y in range(1,4)]
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3), (4, 1), (4, 2),
(4, 3)]
>>> [(x,y) for x in range(1,5) for y in range(1,4) if x != y]
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2), (4, 1), (4, 2), (4, 3)]
>>> [(x,y) for x in range(1,5) for y in range(1,4) if x != y if x%y!=0]
[(1, 2), (1, 3), (2, 3), (3, 2), (4, 3)]
```

c) Les chaînes de caractères

Une chaîne de caractères est une donnée de type `str`. Les chaînes de caractères sont des séquences de caractères individuels, elles se comportent comme des listes (on retrouve les mêmes opérations, la même façon d'utiliser l'indexation).

```
>>> machaine = "bonjour"
>>> machaine[0]
'b'
>>> machaine[6]
'r'
>>> len(machaine)
7
>>> 'o' in machaine
True
>>> 'z' in machaine
False
```

`machaine[k]` est le caractère d'indice `k` dans `machaine`
`machaine[p:n:q]` permet d'extraire une sous-chaîne d'une chaîne

Mais...

```
>>> machaine[1]="t"
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> machaine.append('!')
Traceback (most recent call last):
  File "<console>", line 1, in <module>
AttributeError: 'str' object has no attribute 'append'
```

Contrairement aux listes, **les chaînes de caractères ne sont pas modifiables** : on dit qu'une *chaîne de caractère* est **immutable** tandis qu'une *liste* est **mutable**. Avec une chaîne, on ne peut pas ajouter d'élément avec `append`, ni supprimer un élément avec `pop` ou `remove`, ni modifier les éléments un par un en leur affectant une nouvelle valeur.

Pour **"modifier" une chaîne** de caractères, on doit **remplacer entièrement le contenu de la variable** par une nouvelle chaîne de caractères :

```
>>> machaine = "au revoir"
```

On peut obtenir la liste des fonctions (méthodes) du type `str` avec `dir`. Puis on peut tester certaines de ces fonctions dans l'interpréteur (à essayer) :

```
>>> dir(str)
>>> help(str)
>>> exemple = "il fait beau et chaud"
>>> len(exemple)
>>> max(exemple)
>>> exemple.count("a")
>>> exemple.count("z")
>>> exemple.count("b")
>>> exemple.find("b")
>>> "bonjour".count("o")
>>> "bonjour".count("z")
>>> "bonjour".find("o")
>>> "bonjour".find("z")
>>> exemple.split( )
>>> exemple.split("a")
>>> exemple.append(".") # renvoie une erreur
```

d) Le tuple

Le type `tuple` fonctionne exactement comme une liste, avec des parenthèses à la place des crochets, voir même sans mettre de parenthèses. Par contre, il n'est pas modifiable : le type `tuple` est **immutable**. Il est utilisé implicitement en de nombreux endroits par Python.

```
>>> t1 = (1,4,"toto")
>>> t2 = 42,"pi",11.78,"Python"
>>> t3 = ()
>>> t4 = ("un seul élément",)
```

e) Autres types itérables

Le type `dict` (dictionnaire) et le type `set` (ensemble) sont d'autres types itérables. Nous manipulerons les dictionnaires en TP.

e.1) Dictionnaire

Le type `dict` permet de stocker des collections d'associations **clé→valeur**, et fournissant un accès très rapide à la valeur à partir de la clé. L'itération sur un dictionnaire travaille sur les clés. L'ordre n'est pas défini et peut varier au cours de l'évolution du contenu.

```
>>> d = { "Ain": 1, "Ardèche": 7, "Calvados": 14, "Lozère": 48, "Orne": 61, "Paris": 75,
"Essonne": 91}
>>> d["Paris"]
75
>>> list(d.keys())
['Ardèche', 'Orne', 'Essonne', 'Ain', 'Lozère', 'Paris', 'Calvados']
>>> list(d.values())
[7, 61, 91, 1, 48, 75, 14]
>>> list(d.items())
[('Ardèche', 7), ('Orne', 61), ('Essonne', 91), ('Ain', 1), ('Lozère', 48), ('Paris',
75), ('Calvados', 14)]
>>> for dept in d:
...     print("Département", dept, "code", d[dept])
...
Département Ardèche code 7
Département Orne code 61
Département Essonne code 91
Département Ain code 1
Département Lozère code 48
Département Paris code 75
Département Calvados code 14
```

e.2) Ensemble

Le type `set` permet de stocker des collections de valeurs en offrant des opérations ensemblistes (appartient, inclus, intersection, union, différence...). L'ordre n'est pas défini et peut varier au cours de l'évolution du contenu.

```
>>> s1 = {'A', 'B', 'C', 'D', 'E', 'F'}
>>> s2 = {'E', 'F', 'G', 'H'}
>>> 'C' in s1
True
>>> 'X' in s1
False
>>> s1 & s2
{'F', 'E'}
>>> s1 | s2
{'G', 'F', 'A', 'C', 'D', 'B', 'E', 'H'}
>>> s1 - s2
{'A', 'D', 'B', 'C'}
>>> s1 ^ s2
{'D', 'C', 'G', 'B', 'A', 'H'}
```

e.3) Erreurs

Remarque : les types `int`, `float`, `bool` **ne sont pas itérables**. Exemples :

```
>>> for k in 512 :
...     print(k)
...
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'int' object is not iterable
```

Une **erreur courante**, oublier le `range()` lors d'une itération sur les index d'une liste :

```
>>> lst = [1,5,7,9]
>>> for index in len(lst):
...     print(index, lst[index])
...
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'int' object is not iterable
```


VII - L'instruction de boucle while

1) Principe

C'est une *instruction composée*. Elle permet de **répéter un bloc d'instructions tant qu'une condition reste vraie**.

La syntaxe est :

```
while condition :
    bloc d'instructions secondaire
```

Exemple 1 :

```
1. x = 3
2. while x <= 20 :
3.     x = x+5
4. print(x)
```

Tableau de suivi :

N° de ligne	x	$x \leq 20$	Affichages
1	3		
2		Vrai	
3	8		
2		Vrai	
3	13		
2		Vrai	
3	18		
2		Vrai	
3	23		
2		Faux	
4			23

Définition : le nombre de passages dans une boucle s'appelle le **nombre d'itérations**.
Ce nombre peut être nul... ou potentiellement infini (« boucle sans fin »).

Exemple d'une boucle où on ne passe jamais (zéro itération) :

```
1. x = 50
2. while x <= 20 :
3.     x = x+5
4. print(x)
```

Exemple d'une boucle d'où on ne sort jamais (boucle sans fin) :

```
1. x = 50
2. while x > 10 :
3.     x = x + 1
4. print(x)
```

On peut **compter le nombre d'itérations** avec une variable `compteur` qu'on initialise à 0 *avant d'entrer dans la boucle* :

```
1. x = 3
2. compteur = 0
3. while x <= 20 :
4.     x = x+5
5.     compteur = compteur + 1
6. print("le nombre est égal à", x, "et il y a eu", compteur, "passages dans la boucle.")
```

Tableau de suivi :

n° ligne	x	compteur	$x \leq 20$	Affichages
1	3			
2		0		
3			Vrai	
4	8			
5		1		
3			Vrai	
4	13			
5		2		
3			Vrai	
4	18			
5		3		
3			Vrai	
4	23			
5		4		
3			Faux	
6				le nombre est égal à 23 et il y a eu 4 passages dans la boucle.

Exemple 2 : calcul de la factorielle d'un entier n, première version

```

1. n = 4      # ou n'importe quelle autre entier positif
2. facto = 1
3. compteur = 1
4. while compteur <= n :
5.     facto = facto * compteur
6.     compteur = compteur + 1
7. print("la factorielle de", n, "est", facto)

```

Tableau de suivi (à remplir) :

n° ligne	n	facto	compteur	compteur≤n

Affichage final :

Nombre d'itérations :

Testons ce programme pour n = 0

```

1. n = 0
2. facto = 1
3. compteur = 1
4. while compteur <= n :
5.     facto = facto * compteur
6.     compteur = compteur + 1
7. print("la factorielle de", n, "est", facto)

```

Tableau de suivi (à remplir) :

n° ligne	n	facto	compteur	compteur≤n

Affichage final :

Nombre d'itérations :

2) Difficultés

Deux types de difficultés :

- **le résultat du traitement peut être faux.**

Pour éviter cela, il faut écrire avec soin la condition d'entrée dans la boucle, ainsi que les instructions du bloc secondaire.

Il faut connaître les valeurs des variables à chaque passage dans la boucle ainsi qu'à la sortie de la boucle, afin d'obtenir un résultat correct en fin de traitement.

- **ou bien on peut se trouver « coincé » dans une boucle infinie.**

Pour pouvoir sortir de la boucle, il faut que quelque chose, dans le bloc d'instructions secondaire, amène la condition d'entrée à devenir fausse.

a) Éviter les résultats faux à la sortie de la boucle

Programme exemple 2, version 2... RÉSULTAT FAUX :

```
1. n = 4
2. facto = 1
3. compteur = 1
4. while compteur < n :      # on modifie la condition
5.     facto = facto * compteur
6.     compteur = compteur + 1
7. print("la factorielle de", n, "est", facto)
```

Tableau de suivi (à remplir) :

n° ligne	n	facto	compteur	compteur ≤ n

Programme exemple 2, version 3... RÉSULTAT FAUX :

```
1. n = 4
2. facto = 1
3. compteur = 1
4. while compteur <= n :
5.     # on incrémente le compteur avant le calcul de facto
6.     compteur = compteur + 1
7.     facto = facto * compteur
8. print("la factorielle de", n, "est", facto)
```

Tableau de suivi (à remplir) :

n° ligne	n	facto	compteur	compteur ≤ n

b) Éviter les boucles infinies

Quelques exemples de boucles infinies...

Programme exemple 3 :

```
1. p = 1
2. while p != 0 :      # tant que p est différent de 0
3.     print("coucou !")
```

Tableau de suivi (à remplir) :

n° ligne	p	p ≠ 0

Programme exemple 4 :

```
1. var1 = 3
2. var2 = 4
3. while var1 < 100 :
4.     var2 = var2 * var1
5.     print(var2, var1)
```

Tableau de suivi (à remplir) :

n° ligne	var1	var2	var1<100

Programme exemple 5 :

```
1. var1 = 1
2. while var1 != 99 :
3.     var1 = 2 * var1
```

Tableau de suivi (à remplir) :

n° ligne	var1	var1≠99

Programme exemple 6 :

```
1. var1 = 1
2. while var1 <= 99 :
3.     var1 = var1 - 1
```

Tableau de suivi (à remplir) :

n° ligne	var1	var1≠99

Programme exemple 2, version 4 :

```
1. n = 4 # ou n'importe quelle autre entier positif
2. facto = 1
3. compteur = 1
4. while compteur <= n :
5.     facto = facto * compteur
6. print("la factorielle de", n, "est", facto)
```

Tableau de suivi (à remplir) :

n° ligne	n	facto	compteur	compteur≤n

b.1) Conseils & astuces

Pour essayer d'éviter les **boucles infinies**, un prérequis est d'**identifier les variables de condition** (ce sont les variables présentes dans l'expression de la condition de boucle) et de s'assurer qu'à chaque itération au moins une de ces variables est modifiée¹⁸.

Ensuite, il faut **bien analyser l'évolution des variables de condition** afin de vérifier qu'elles atteindront bien à un moment une valeur permettant à la condition de boucle de devenir fausse.

Si un programme script Python entre dans une boucle infinie... vous pouvez essayer de l'interrompre avec :

- Sous **IDLE**, **Ctrl-C** (assez général, signal envoyé au programme pour qu'il s'interrompe)

¹⁸ - Il peut arriver que les modifications qui permettent à l'expression de la condition de boucle ne proviennent pas de la logique du bloc, mais d'événements extérieurs (écoulement du temps, changement d'état dans le système géré par ailleurs).

- Sous **Pyzo**, **Ctrl-K** (redémarre le shell d'exécution du script Python, Ctrl-Maj-K pour terminer le shell sans le redémarrer)

3) Boucle while ou boucle for ?

En principe, on doit utiliser :

- une boucle **while** si le **nombre d'itérations n'est pas connu à l'avance** ;
- une boucle **for** si le **nombre d'itérations est connu à l'avance** (cela inclut donc les boucles sur les séquences)

Note : en conclusion, l'exemple 2 de calcul de la factorielle d'un entier aurait dû être écrit avec une boucle **for** ! On est sûr que la boucle se termine et, à condition de bien gérer les bornes, on a un algorithme moins sujet à erreurs de logique.

VIII - Les fonctions : créer des sous-programmes

Afin d'écrire nos programmes de façon plus concise, nous pouvons définir nos propres fonctions pour pouvoir les appeler ensuite, tout comme nous avons pris l'habitude d'appeler des fonctions prédéfinies déjà existantes.

On dira que l'on **définit** une nouvelle fonction, lorsqu'on nomme par un nouveau nom, un bloc d'instructions (éventuellement en y faisant intervenir des variables qu'on appellera paramètres).

On dira que l'on **appelle** cette fonction lorsqu'on saisit son nom dans l'interpréteur ou dans un script (programme) python.

Ainsi, les programmeurs du module `math` ont *défini* pour nous (c'est à dire programmé) les fonctions `sin()`, `cos()`, `sqrt()`, etc, pour que nous puissions ensuite les *appeler* en saisissant par exemple :

```
var = sin(pi)
print(cos(0))
x = sqrt(2)
# etc.
```

Nous allons maintenant apprendre à définir, nous aussi, de nouvelles fonctions.

1) Définir une fonction

La définition d'une fonction commence par une **ligne d'introduction** qui doit contenir : le **mot clé `def`**, suivi du **nom de la fonction**, suivi de tous les **paramètres** de la fonction entre parenthèses.

Après cette ligne d'introduction, les instructions qui définissent la fonction doivent être dans un **bloc indenté** qui constitue le **corps de la fonction** (si une instruction n'est pas indentée : elle sera considérée comme ne faisant pas partie de la définition de la fonction et entraînera la fin de la définition du corps de la fonction).

Le mot clé **`return`** permet d'indiquer la **valeur retournée** par la fonction. Il achève la définition de la fonction (lorsque Python rencontre un **`return`** dans une fonction, *il sort de la fonction*, les instructions qui pourraient se trouver ensuite dans le corps de la fonction seront ignorées).

```
def nomDeMaFonction(parametre1, parametre2, parametre3):
    """docstring
    documentation de la fonction qui s'affichera en tapant
    help(nomDeMaFonction)
    """
    ...bloc d'instructions définissant la fonction...
    return valeur_retournee
```

La valeur retournée indiquée après le mot clé **`return`** est le **résultat** du « travail » de la fonction. Tout ce qui a pu être calculé localement dans la fonction est perdu s'il n'est pas retourné.

Si on ne met **pas de `return`** dans le corps de la fonction, ou bien un **`return` sans spécifier de valeur**, alors Python retourne automatiquement la valeur **`None`**.

À l'IUT, pour être explicite, si une fonction ne retourne rien, on écrira systématiquement : **`return None`**.

2) Exemples

Nous allons **définir** ci-dessous 6 fonctions différentes.

Exemple 1 :

```
def f(x) :
    """ cette fonction calcule 3*x+1 et retourne le résultat """
    resultat = 3*x+1
    return resultat
```

Exemple 2 :

```
def sommeCarres(param1 = 3, param2 = 4) :
    """ cette fonction calcule la somme des carrés de
    ses 2 arguments, et retourne le résultat """
    calcul = param1 ** 2 + param2 ** 2
    return calcul
```

Exemple 3 :

```
def salutation(param_prenom) :
```

```

""" cette fonction affiche une formule de bienvenue
et ne retourne rien"""
print("bonjour",param_prenom)
return None

```

Remarque : ne pas confondre la fonction `print()` et le **mot-clé** `return`. Ce n'est pas parce qu'une fonction affiche quelque chose à l'écran qu'elle retourne une valeur. Et le mot clé `return` ne provoque pas d'affichage à l'écran (l'affichage dans le cadre d'un Shell Python est réalisé par l'environnement du shell qui automatiquement affiche le résultat des expressions évaluées - sauf si ce résultat est `None`).

Exemple 4 :

```

def factorielle(param) :
    """ cette fonction calcule la factorielle de son argument
    et retourne le résultat"""
    facto = 1
    for k in range(1, param+1):
        facto = facto*k
    return facto

```

Exemple 5 :

```

def avance_et_traceCercle() :
    """ cette fonction avance de 50 px avec la tortue, puis
    trace un cercle de rayon 100 px. Elle retourne la position finale
    de la tortue.
    """
    from turtle import forward, circle, position
    forward(50)
    circle(100)
    return position()

```

Exemple 6 :

```

def produitScalaire(p_vect1, p_vect2) :
    """ cette fonction calcule le produit scalaire (en repère
    orthonormé) de 2 vecteurs dont les coordonnées sont passées en
    paramètre sous forme de listes. Elle retourne le résultat"""
    prod = 0
    for k in range(len(p_vect1)):
        prod = prod + p_vect1[k]*p_vect2[k]
    return prod

```

Exemple 7 :

```

def div_euclidienne(p_nb1, p_nb2) :
    quotient = p_nb1 // p_nb2
    reste = p_nb1 % p_nb2
    return quotient, reste

```

Cette fonction retourne 2 valeurs séparées par des virgules. Il s'agit en fait d'un `tuple` dont les parenthèses ont été omises.

3) Le programme principal

Les instructions qui ne font *pas partie des blocs de définitions de fonctions* sont appelées le **programme principal**. Généralement on regroupe entre elles les instructions du programme principal en les écrivant en fin de module, après toutes les définitions de fonction.

Exemple 8 :

<pre> 1. #!/usr/bin/python3 2. # -*- coding: UTF-8 -*- 3. """ 4. (DocString) Ce module détermine les résultats semestriels des étudiants. 5. """ 6. # fichier: resultats_semestre.py 7. # auteur : prof 8. # date : septembre 2014 </pre>	<p>(1) On écrit d'abord un en-tête de module, constitué d'une chaîne de documentation (affiché avec <code>help(module)</code>), et de méta-informations (date, auteur...).</p>
<pre> 9. ### imports ### 10. 11. import random 12. import turtle </pre>	<p>(2) On rassemble tous les imports en début de module.</p>

<pre> 13. ### constantes et variables globales ### 14. 15. NOTE_MIN_UE = 8 16. MIN_MOYENNE = 10 17. COEF_UE1 = 10 18. COEF_UE2 = 9 19. COEF_UE3 = 11 </pre>	<p>(3) Puis on indique toutes les déclarations (initialisations) de <i>constantes</i> et de <i>variables globales</i>.</p>
<pre> 20. ### définitions ### 21. 22. def resultat(p_unite1, p_unite2, p_unite3) : 23. """détermine et retourne le résultat du semestre""" 24. somme = COEF_UE1 + COEF_UE2 + COEF_UE3 25. moy=(p_unite1*COEF_UE1+p_unite2*COEF_UE2+ 26. p_unite3* COEF_UE3) /somme 27. return moy 28. 29. def graphique(p_reussi): 30. if p_reussi : 31. turtle.color('black', 'green') 32. turtle.begin_fill() 33. turtle.circle(100) 34. else : 35. turtle.color('black', 'red') 36. turtle.begin_fill() 37. turtle.circle(100) 38. turtle.end_fill() 39. return None </pre>	<p>(4) On définit toutes les fonctions du module, les unes à la suite des autres.</p>
<pre> 40. ### programme principal ### 41. 42. prenom = input("quel étudiant ? ") 43. print("voici les résultats de", prenom, ":") 44. 45. note1 = random.randrange(21) 46. note2 = random.randrange(21) 47. note3 = random.randrange(21) 48. print("Les notes des UE 1 2 et 3 sont", note1, note2, note3) 49. 50. moyenne_generale = resultat(note1, note2, note3) 51. print("La moyenne générale est de", moyenne_generale) 52. 53. valide = note1>=NOTE_MIN_UE and note2>=NOTE_MIN_UE and 54. note3>=NOTE_MIN_UE and moyenne_generale>=MIN_MOYENNE 55. 56. graphique(valide) </pre>	<p>(5) On écrit enfin le programme principal du module.</p>
<pre> 56. """ 57. >>> 58. quel étudiant ? Georges 59. voici les résultats de Georges : 60. - les notes des UE 1 2 et 3 sont 0 13 13 61. - la moyenne générale est de 8.666666666666666 62. >>> ===== RESTART 63. ===== 64. >>> 65. quel étudiant ? Josette 66. voici les résultats de Josette : 67. - les notes des UE 1 2 et 3 sont 10 16 7 68. - la moyenne générale est de 10.7 69. >>> ===== RESTART 70. ===== 71. >>> 72. quel étudiant ? Gaston 73. voici les résultats de Gaston : 74. - les notes des UE 1 2 et 3 sont 10 18 20 75. - la moyenne générale est de 16.066666666666666 76. >>> 77. """ </pre>	<p>(6) À l'IUT, en fin de module, on copie-colle les résultats d'exécution entre triple-guillemets.</p>

4) Paramètres d'une fonction

L'utilisation de paramètres dans la définition d'une fonction permet de donner à cette fonction un caractère de généralité. Les paramètres correspondent à des variables dont les valeurs seront fixées lors de l'appel de la fonction.

Si l'on reprend nos exemples précédents :

Exemple 1 : la fonction `f(x)` a un seul paramètre qui est `x`.

Ainsi, on pourra calculer et retourner la valeur de l'expression $3 \times x + 1$, quelle que soit la valeur de `x`.

Exemple 2 : la fonction `sommeCarres(param1, param2)` a deux paramètres qui sont `param1` et `param2`. Ainsi, cette fonction est capable de calculer et retourner la valeur de l'expression $a^2 + b^2$, pour `a` et `b` quelconques.

Exemple 5 : la fonction `avance_et_traceCercle()` n'a *aucun paramètre*. Elle ne peut rien faire d'autre qu'avancer de 100 pixels puis tracer un cercle de rayon 50 pixels.

👉 Les paramètres d'une fonction n'ont aucune signification en dehors du corps de cette fonction (ce bloc est appelé la **portée locale** de la fonction). Ils n'existent pas hors de la fonction.

5) Appeler une fonction

a) Les arguments remplacent les paramètres

Tout comme cela est indiqué dans le paragraphe IV - 4 (page 25), pour appeler une fonction il suffit d'indiquer le nom de cette fonction, suivi de ses **arguments** entre parenthèses : pour que la fonction puisse effectuer sa tâche et retourner le résultat souhaité, **les valeurs des arguments vont être affectées aux paramètres** et ainsi être utilisées dans les instructions du corps de la fonction lors de leur exécution.

Ces arguments **passés lors de l'appel** sont donc les valeurs particulières sur lesquelles on demande à la fonction de travailler.

Exemple :

```
print("la factorielle de 4 est", factorielle(4))
var=factorielle(5)
print("la factorielle de 5 est", var)
res = sommeCarres(2,3)
salutation("sabine")
nombre = input("de quel nombre souhaitez vous calculer la factorielle ? ")
print("la factorielle de",nombre,"est", factorielle(nombre))
```

Lorsque la fonction **retourne plusieurs valeurs** (ceci est toujours fait sous forme d'un tuple), on accède aux différents éléments soit *par leur indice* ou bien en réalisant une *affectation à de multiples variables* (autant qu'il y a de valeurs retournées).

```
>>> res = div_euclidienne(20,3)
>>> res[0]
6
>>> res[1]
2
>>> quotient,reste = div_euclidienne(20,3)
>>> quotient
6
>>> reste
2
```

Remarques :

- 1) Ne pas confondre « arguments » et « paramètres » d'une fonction. Les paramètres font partie de la définition de la fonction. Les arguments sont les valeurs fournies à la fonction lors d'un appel, cela peut-être une valeur littérale, une variable, ou toute expression fabriquant une valeur.

Note : dans certaines langues on parle de *paramètres actuels* pour les arguments et de *paramètres formels* pour les paramètres.

- 2) Les paramètres d'une fonction n'ont pas de signification hors de la portée locale, le corps de cette fonction. Ils est impossible d'accéder à ces paramètres (par leur nom), en dehors du bloc de la fonction (cela génère une erreur). À l'IUT, par convention, on nomme souvent les paramètres avec un préfixe

p_... afin de les identifier, et il est interdit d'utiliser les mêmes noms pour les variables du programme principal.

b) Ordre des arguments

Les arguments passés lors de l'appel doivent figurer entre les parenthèses **dans le même ordre** que celui des paramètres dans la définition de la fonction.

Exemple : reprenons une des fonctions de l'exemple 8 : la fonction `resultat()` qui permet de calculer le résultat d'un semestre à partir des notes des 3 UE est définie de la façon suivante :

```
def resultat(p_unite1, p_unite2, p_unite3):
```

Lorsqu'on appelle cette fonction, on doit donc indiquer les notes des 3 UE *dans le même ordre* que celui qui a été prévu dans cette définition, c'est à dire : UE1 puis UE2 puis UE3 :

```
>>> resultat(15, 9, 11)
```

Une dérogation à cette règle : lorsqu'on appelle une fonction, on peut adopter un ordre quelconque pour les arguments *lorsqu'on précise les noms* des paramètres associés pour chaque argument¹⁹. Exemple :

```
>>> resultat(p_unite2 = 9, p_unite1 = 15, p_unite3=11)
```

c) Paramètres "optionnels" avec une valeur par défaut

Certains paramètres peuvent être définis avec une **valeur par défaut**. Il devient alors *facultatif* d'indiquer leur valeur au moment de l'appel de la fonction, la valeur par défaut étant utilisée à la place de l'argument manquant.

Exemple 1 : la fonction `print` peut prendre autant d'arguments qu'on veut (ils sont optionnels). Elle a un argument optionnel `end` dont la valeur par défaut est `"\n"`, c'est à dire un caractère invisible de retour à la ligne.

```
>>> print("bonjour", end="!!")
bonjour!!
>>> print("bonjour")           #ici le paramètre end n'est pas précisé, mais il existe et sa valeur est "\n"
```

Exemple 2 :

```
>>> sommeCarres(0,5)
>>> sommeCarres(0)
>>> sommeCarres()
>>> sommeCarres(param2=7)
```

Exemple 5 : modifions la fonction `avance_et_traceCercle()` de la façon suivante :

```
def avance_et_traceCercle(p_avant = 50, p_rayon = 100) :
    from turtle import forward, circle, position
    forward(p_avant)
    circle(p_rayon)
    return position()

>>> avance_et_traceCercle()
(50.00,-0.00)
>>> avance_et_traceCercle(10,400)
(60.00,0.00)
>>> avance_et_traceCercle(-40)
(20.00,0.00)
>>> avance_et_traceCercle(p_rayon = 70)
(70.00,0.00)
```

6) Portée des variables

Chaque nom de variable est connu dans un **"espace de nom"** et inconnu hors de cet espace. Cet espace de nom est aussi appelé **"portée locale"** de la variable.

Par exemple la constante `pi` est connue dans le module `math`, et inconnue en dehors. Si on souhaite l'utiliser dans un module (appelé aussi programme ou script) python, il faut l'importer pour qu'elle soit connue :

```
from math import pi
print("la valeur de pi est", pi)
```

¹⁹ - En fait, on peut mixer arguments positionnels et arguments nommés, en mettant les premiers d'abord afin qu'ils soient associés aux paramètres de la déclaration dans le même ordre.

D'une manière générale, *les variables sont inconnues hors de leur portée locale*, c'est à dire hors du bloc où elles ont été affectées *pour la première fois*. Ainsi :

- Les variables déclarées dans le bloc d'instructions d'une fonction sont inconnues hors de la portée locale de cette fonction (hors du corps de cette fonction).
- Les paramètres des fonctions sont inconnus hors de la portée locale de cette fonction (hors du corps de cette fonction).
- Les variables déclarées dans les instructions d'un module sont inconnues hors de la portée locale de ce module.

Une variable qui est connue dans tout un module est appelée **variable globale**. Une variable qui n'est connue que dans une fonction (c'est à dire dans le bloc d'instructions du corps de cette fonction) est appelée **variable locale**.

Dans l'exemple 1 précédent, la variable `resultat` est une variable locale de la fonction `f()`. Dans l'exemple 2, la variable `calcul` est une variable locale à la fonction `sommeCarres()`. Dans l'exemple 7, les variables `quotient` et `reste` sont des variables locales à la fonction `div_euclidienne()`.

Remarque : ne pas confondre les *paramètres* et les *variables locales*. **Les paramètres ne doivent pas être redéfinis dans le corps de la fonction** (l'appel à la fonction fournit les valeurs en arguments pour les paramètres, aucune raison de redemander une valeur à l'utilisateur ou de changer arbitrairement de façon systématique une valeur). Exemple à ne pas suivre :

```
def afficheAccents(p_nbCaract,p_nblignes):
    p_nbCaract = input("combien de caractères ? ") # 4 NON 4
    p_nblignes = 15                               # 4 NON 4
    return None
```

Afin d'éviter les confusions, à l'IUT on adoptera la convention suivante :

- les paramètres des fonctions seront nommés par un nom débutant par `p` ou `p_`
- les variables globales pourront être nommées par un nom débutant par `g_`
- on ne donnera jamais le même nom aux paramètres des fonctions et aux arguments passés lors de l'appel de ces fonctions.
- si on veut éviter les confusions, on pourra utiliser des noms différents pour les variables locales des différentes fonctions ainsi que pour les variables globales du module.

👉 Pour les fonctions, penser « **boîte noire** » : considérez que la fonction est entourée d'un mur avec deux passages : les *paramètres* qui permettent de faire *entrer* des valeurs dans la fonction, et le *return* qui permet de faire *sortir* le résultat de la fonction.

L'accès aux paramètres de la fonction et aux variables locales à la fonction n'est pas possible hors du corps de la fonction (donc ni du programme principal, ni des autres fonctions).

L'accès aux variables globales définies au niveau du programme principal est à éviter autant que possible (sauf pour les constantes).
(voir Portée des variables page 49)

7) Fonctions récursives

Une fonction peut s'appeler elle-même. Une telle fonction est appelée **fonction récursive**.

Exemple :

```
def factorielle(p_nombre):
    if nombre > 0:
        return p_nombre*factorielle(p_nombre - 1)
    else :
        return 1
```

Elle est typiquement utilisée lorsqu'une expression dans la fonction nécessite un résultat qui peut être produit par un appel à la fonction elle-même.

Il est important dans ce genre de fonctions de s'assurer qu'il y a **une condition permettant de stopper les appels récursifs**, sinon on peut entrer dans une récursion sans fin (de la même façon qu'on peut avoir des boucles **while** sans fin).

8) Méthodes

Les fonctions qui s'appliquent à des objets avec une notation `variable.fct()` s'appellent des **méthodes**. Ces notions sont liées à la programmation orientée objet, traitée plus spécifiquement au chapitre XII (page 65).

Exemples : `append()`, `pop()`, `upper()` sont des méthodes.

```
>>> l1 = ["titi","tata"]
>>> l1.append("toto")
>>> l1
['titi', 'tata', 'toto']
>>> l1.pop()
'toto'
>>> l1
['titi', 'tata']
>>> machaine = "Hello"
>>> machaine.upper()
'HELLO'
```


IX - Organisation des applications en plusieurs modules

Un **module** Python est simplement un **fichier script Python**. La base du nom du fichier, **sans l'extension .py**, constitue le **nom** du module. Ce nom doit respecter les *règles des identificateurs* Python (comme les noms de variables, les noms de fonctions), et donc ne peut contenir que des caractères alphabétiques, numériques et le souligné (`_`). Il ne peut pas commencer par un caractère numérique, il ne peut pas contenir d'espace, et on évitera les caractères accentués.

On regroupe dans des modules outils les fonctions *par thèmes*. On **importe** ensuite ces modules outils dans les modules qui nécessitent d'utiliser leurs fonctions. Le programme (ou "application") est ainsi mieux organisé.

a) Définition d'un module

Un module reprend la même structure que celle décrite dans Le programme principal page 46, nous ne la décrivons pas de nouveau ici.

Dans chaque module définissant des fonctions, on peut écrire un **programme principal de test du module** permettant de vérifier le bon fonctionnement des différentes fonctions du module. Ceci permet d'avoir directement un code de test que l'on peut ré-exécuter dès que l'on fait une modification pour vérifier que l'on n'a pas introduit d'erreur (cela peut aussi servir d'exemple d'utilisation des fonctions définies dans le module).

Pour éviter que le programme principal de test du module ne soit exécuté lorsque le module est importé comme module outil par un autre module, le bloc d'instructions de ce programme principal de test est placé dans une condition **if** spéciale²⁰ :

```
if __name__ == "__main__" :
    # Programme principal de test:
    res = factorielle(3)
    if res != 6:
        print("factorielle(): Erreur")
    else:
        print("factorielle(): Ok")
```

afin de ne s'exécuter que lorsque c'est lui même qui est appelé (et pas lorsqu'il est importé par un autre module).

b) Utilisation d'un module

On **importe** généralement un module *au début d'un programme* (ou d'un autre module), afin de rendre accessible les noms de variables ou fonctions qui y sont définis.

```
import math
import monmodule
import sys
```

On peut ensuite **accéder aux noms** (variables, fonctions) définis dans le module importé en utilisant la notation `nommodule.nomvariable` ou `nommodule.nomfonction`.

```
x = math.cos(1.705 + math.pi)
y = math.sin(1.705 + math.pi)
valeur = monmodule.factorielle(34)
arguments = sys.argv
```

Pour tout ce qui est fonction ou constante (variables dont la valeur ne doit pas changer), on peut **importer directement les noms désirés**. Ces noms sont ensuite utilisables sans les préfixer le nom du module. Si on reprend l'exemple ci-dessus :

```
from math import cos, sin, pi
from monmodule import factorielle
import sys
x = cos(1.705 + pi)
y = sin(1.705 + pi)
```

20 - Attention, erreurs courantes : il y a deux `_` accolés de part et d'autres dans la variable `__name__` et dans la chaîne `'__main__'`. Et il y a bien un espace entre le **if** et le `__name__`.

```
valeur = factorielle(34)
arguments = sys.argv
```

👉 **Attention** (sources de bugs) :

- 1) Il est *fortement déconseillé d'importer directement les variables* des autres modules (hors constantes). On crée en effet dans le module d'import une seconde variable de même nom, qui initialement référence les mêmes données... mais dont la valeur pourrait dans le temps diverger de celle de la variable d'origine.
- 2) Il existe une notation `from module import *`, qui permet d'un seul coup d'importer tous les noms définis dans un module. Cela peut être pratique pour des tests rapides, par exemple dans le Shell Python. Mais **cela NE DOIT PAS être utilisé dans des scripts**, on ne sait en effet pas toujours ce que l'on importe ainsi, (l'auteur du module importé pourrait dans une nouvelle version y ajouter de nouveaux noms ou en supprimer), et on prend le risque d'avoir des **conflits de noms**.

Erreur courante : `from os import *` crée un conflit de nom en masquant la fonction *standard* `open()` par celle *spécifique* définie dans le module `os` et qui a un *comportement différent*.

c) Renommage de noms importés

Lors de l'importation, on peut renommer un nom importé avec `as` :

```
import turtle as tortue
from math import sqrt as racine
```

Cela permet d'utiliser des noms plus courts dans certains cas, de résoudre les problèmes lorsque deux modules importés définissent le même nom, etc.

X - Les fichiers

On peut stocker des informations dans des fichiers enregistrés sur le disque dur, sur une clé usb, etc. Ces fichiers peuvent contenir tout type de données (toujours sous forme de séquence d'octets), pour notre apprentissage on se limitera aux fichiers ne contenant que du texte. Les fichiers texte se terminent généralement par .txt, mais on retrouve des fichiers textes avec de nombreuses autres extensions : .dat, .ini, .cfg, .html, .xml...).

Pour travailler avec un fichier, il faut :

- 1) ouvrir un *fichier existant* en lecture ou en écriture, ou créer un *nouveau fichier*
- 2) lire les données du fichier, ou écrire des données dans le fichier
- 3) fermer le fichier

1) Ouvrir un fichier

On utilise la fonction standard `open()` :

```
varfichier = open(nom_du_fichier, mode_d'accès, encoding=encodage)
```

Les paramètres définissent :

- *nom_du_fichier* : une chaîne qui permet de localiser et d'identifier le fichier sur le disque (voir Organisation des fichiers sur disque, page 59).
- *mode_d'accès* : un chaîne qui spécifie les opérations que l'on veut faire avec le fichier : 'r' pour la lecture (read), 'w' pour l'écriture (write), 'a' pour l'ajout (append)...²¹
- Qui n'a pas déjà vu apparaître sur une page web le texte 'Ã©' à la place d'un 'é' ? C'est un problème d'encodage/décodage de texte.

encodage : (paramètre nommé `encoding`, optionnel... mais fortement recommandé ! ²²) une chaîne qui indique de quelle façon les octets qui sont dans le fichiers sont codés pour correspondre aux caractères en mémoire. On utilise maintenant de préférence 'utf-8'. Si on ne stocke que des nombres et du texte sans accent, l'encodage 'ascii' suffit (il est compatible avec 'utf-8'). On trouve aussi souvent des fichiers en 'latin1', ou en 'cp1252' sous Windows, car ces deux formats permettent de représenter les caractères accentués du français.

Exemples :

```
les_etudiants = open("etudiants.txt", "r", encoding = "utf-8")
les_resultats = open("appreciations.txt", "w", encoding = "utf-8")
les_notes = open("notes.txt", "a")
```

La fonction `open()` ainsi appelée construit et renvoie dans la variable `varfichier`, un "objet-fichier texte" qui va gérer la communication (« entrées/sorties ») entre le programme python et le véritable fichier (support physique) sur le disque dur, par l'intermédiaire des *librairies Python* et du *système d'exploitation*.

```
>>> varfichier = open("unfichier.txt", "w", encoding="utf-8")
>>> type(varfichier)
<class '_io.TextIOWrapper'>
```

C'est cette variable `varfichier` qu'on va utiliser lors des appels aux méthodes spécifiques aux fichiers. On écrira : `varfichier.nom_de_la_methode(arguments)`.

On trouve aussi des attributs du fichier par cette variable :

```
>>> varfichier.encoding
'utf-8'
>>> varfichier.mode
'w'
```

21 - Il y a d'autres modes d'ouverture, dont 'r+' pour la lecture/écriture d'un fichier existant, 'a+' pour la lecture/écriture d'un fichier pouvant ne pas exister.

22 - Si on n'a pas utilisé le bon encodage, dans le meilleur des cas on a des erreurs `UnicodeDecodeError` ou `UnicodeEncodeError`, dans le moins bon des cas on travaille sur des données invalides ou enregistre des données invalides.

```
>>> varfichier.name
'unfichier.txt'
>>> varfichier.closed
False
```

2) Fermer un fichier

Lorsque les accès au fichier sont terminés, à la fin des lectures ou des écritures, **il faut le fermer**.

```
varfichier.close()
```

Plusieurs raisons à cette obligation :

- 1) Le système d'exploitation autorise un **nombre limité d'ouverture de fichiers** simultanée pour un programme en cours d'exécution. Si on ouvre des fichiers sans les refermer²³, on peut atteindre ce nombre et le programme n'est alors plus capable d'ouvrir d'autres fichiers.
- 2) Les bibliothèques Python et le système d'exploitation gèrent des **tampons** (« buffers ») pour les lectures et les écritures dans les fichiers. Ceci permet d'adapter la lecture/écriture séquentielle d'un nombre variables d'octets à l'organisation des données sur le disque par blocs de taille fixe. La fermeture du fichier permet de libérer les zones mémoire des tampons, et d'assurer que les dernières données à écrire sont bien transférées dans l'espace de stockage du fichier sur le disque²⁴.

3) Ecrire dans un fichier

Le fichier doit avoir été **ouvert en écriture** avec l'option de mode d'accès **"w"** (write) ou **"a"** (append).

- L'option **"w"** crée un **nouveau fichier vide** du nom choisi. Si le fichier existait déjà, tout son contenu sera effacé.
- L'option **"a"** permet d'**ajouter du contenu** à un fichier sans effacer le contenu déjà présent. Si un fichier de même nom existe déjà, les écritures réalisées sont automatiquement faites à la suite des données existantes.

La méthode pour écrire dans un fichier est `write`, qui ne prend toujours qu'**un seul paramètre** (contrairement à la fonction `print` !²⁵), qui est obligatoirement une chaîne de caractères.

```
varfichier.write("chaîne de caractères")
```

Contrairement à la fonction `print`, la méthode `write` ne génère pas automatiquement de saut de ligne entre deux écritures. Pour aller à la ligne, il faut **écrire explicitement un caractère de retour à la ligne** **"\n"** (on peut le mettre à la fin de la chaîne à écrire). Ce caractère est invisible lorsqu'on ouvre le fichier dans un éditeur, mais il représente un retour à la ligne dans l'éditeur et provoque visuellement le saut de ligne.

Exemple 1 : ouverture avec l'option **"w"** puis écriture

```
monfichier = open("test.txt", "w")
for k in range(100):
    monfichier.write("ligne n° " + str(k)+"\n")
monfichier.close()
```

Exemple 2 : ouverture avec l'option **"a"** puis écriture

```
monfichier = open("test.txt", "a")
for k in range(15):
    monfichier.write("la suite avec le numéro " + str(k)+"\n")
monfichier.close()
```

23 - On peut avoir, avec raison, l'impression que dans certains cas Python referme automatiquement le fichier — c'est le cas lorsque la variable qui référence le fichier disparaît (ex. variable locale d'une fonction lorsque l'on sort de cette fonction). Mais il est très dangereux de s'y fier : certaines implémentations de Python ne fonctionnent pas comme cela, de même que de nombreux autres langages.

24 - On peut forcer le vidage des tampons en écriture sans avoir à refermer le fichier avec un appel à la méthode `varfichier.flush()`.

25 - Il est possible d'utiliser `print` pour écrire dans un fichier, en utilisant un paramètre nommé `file` qui spécifie le fichier dans lequel écrire : `print("x=", x, "y=", y, file=varfichier)`.

Il est possible d'écrire une série de chaînes en utilisant la méthode `writelines` et en lui fournissant une liste de chaînes de caractères. Là encore, la méthode ne fait rien d'automatique pour les fins de lignes, il faut explicitement mettre des `'\n'` dans les chaînes pour qu'ils soient écrits.


```
varfichier.writelines(["liste de", " chaîne de", " caractères"])
```

Exemple 3 : ouverture avec l'option "w" puis écriture avec `writelines`

```
monFichier = open("exemple3.txt", "w", encoding="utf8")
monFichier.writelines(["zzz", "yyy", "xxx", "www", "vvv"])
monFichier.writelines(["zzz\n", "yyy\n", "xxx\n", "www\n", "vvv\n"])
monFichier.close()
```

4) Lire un fichier

Le fichier doit avoir été **ouvert en lecture** avec l'option de mode d'accès `"r"` (read).

 Le fichier contient du **texte** qu'il faudra **convertir en nombres** entiers ou flottants si on veut en faire des graphiques ou les utiliser dans des calculs.

Dans un fichier texte, chaque ligne est délimitée par le caractère de retour à la ligne `"\n"`.

Le principe de lecture du fichier est le suivant : le fichier est lu petit à petit, en partant du début. Un « curseur de lecture » se déplace dans le fichier au fur et à mesure de la lecture, indiquant la position de démarrage pour la lecture suivante. Tout ce qui a déjà été lu ne peut plus être relu (ou alors, il faut fermer le fichier et l'ouvrir à nouveau²⁶).


```
varfichier.read(nombre_de_caractères) ► Renvoie une chaîne de caractères qui contient
autant de caractères du fichier que le nombre de caractères passé en argument.

varfichier.read() ► Renvoie une chaîne de caractères qui contient tout le contenu du fichier.
À ne pas utiliser pour de gros fichiers.

varfichier.readline() ► Renvoie une chaîne de caractères qui contient une ligne du fichier.

varfichier.readlines(nombre_de_lignes) ► Renvoie une liste de chaînes de caractères qui
contient autant de lignes que le nombre de lignes passé en argument.

varfichier.readlines() ► Renvoie une liste de chaînes de caractères qui contient tout le
contenu du fichier découpé par lignes. À ne pas utiliser pour de gros fichiers.
```

 Lors des **lectures par ligne**, chacune des lignes se termine par le caractère de retour à la ligne `"\n"`... sauf éventuellement pour la dernière ligne si le fichier ne se finit pas par un retour à la ligne.

Lorsque la fin du fichier est atteinte, les lectures retournent des chaînes de caractères ou listes vides.

Exemple 4 : lecture avec la méthode `read()`

```
monfichier = open("test.txt", "r")
varcontenu = monfichier.read()
print(varcontenu)
# ou bien on peut écrire directement print(monfichier.read())
monfichier.close()
```

Exemple 4 bis : lecture avec la méthode `read(nombre)`

```
monFichier = open("test.txt", "r", encoding="utf-8")
varcontenu = monFichier.read(5)
print(varcontenu)
varcontenu = monFichier.read(10)
print(varcontenu)
varcontenu = monFichier.read(8)
print(varcontenu)
varcontenu = monFichier.read(5)
print(varcontenu)
```

26 - Ou utiliser la méthode `varfichier.seek(0)` pour remettre le curseur de lecture au début du fichier.

```
# ou bien
print(monFichier.read(5))
print(monFichier.read(5))
monFichier.close()
```

Exemple 5 : lecture avec la méthode `readline()` pour renvoyer une ligne dans une chaîne de caractères

```
monFichier = open("test.txt", "r", encoding="utf-8")
varcontenu = monFichier.readline()
print(varcontenu, end="")
varcontenu = monFichier.readline()
print(varcontenu, end="")
varcontenu = monFichier.readline()
print(varcontenu, end="")
#ou bien
print(monFichier.readline(), end="")
monFichier.close()
```

Exemple 6 : lecture avec la fonction `readlines()` pour renvoyer toutes les lignes dans une liste

```
monFichier = open("test.txt", "r", encoding="utf8")
varcontenu = monFichier.readlines()
print(varcontenu)
for uneLigne in varcontenu :
    print(uneLigne)
monFichier.close()
```

a) Boucle de lecture

En Python un fichier texte peut être considéré comme une **séquence de lignes**, assimilable à une liste de chaînes qui sont fournies au fur et à mesure. On peut donc **itérer sur l'objet-fichier** (c'est à dire l'utiliser dans une boucle `for`). Cette façon de faire est utilisable entre autre pour des gros fichiers qui sont lus par ligne au fur et à mesure et non pas chargés entièrement en mémoire d'un coup.

Exemple 7 : itération sur l'objet fichier

```
monFichier = open("test.txt", "r", encoding="utf8")
print(monFichier)
for uneLigne in monFichier :
    print(uneLigne)
monFichier.close()
```

Là encore, chacune des lignes se termine par le caractère de retour à la ligne `"\n"`... sauf éventuellement pour la dernière ligne si le fichier ne se finit pas par un retour à la ligne.

b) Lecture avec bloc gardé

(en anticipant sur les traitements d'erreurs traité au chapitre XI, page 61)

Pour s'assurer qu'un fichier est bien refermé *même en cas d'erreur*, le code ressemblerait à :

```
monFichier = open("resultats.txt", "w", encoding="ascii")
try:
    for x in donnees:
        res = calculs(x)
        monFichier.write(str(res) + '\n')
finally:
    monFichier.close()
```

Ceci assure que, une fois que le fichier a pu être ouvert (la variable `monFichier` a reçu par affectation l'objet fichier texte issu de l'ouverture), toute terminaison du programme, normale ou anormale (par exemple si un calcul génère une exception liée à une erreur de division par zéro), provoquera bien la fermeture du fichier (grâce à la construction `try/finally`).

L'utilisation d'un « bloc gardé » avec une ressource fichier permet de réduire l'écriture :

```
with open("resultats.txt", "w", encoding="ascii") as monFichier:
    for x in donnees:
        res = calculs(x)
        monFichier.write(str(res) + '\n')
```

Dans le cas d'une boucle de lecture de fichier, on a couramment :

```
with open("data.txt", encoding="utf-8") as f:
    for ligne in f:
        # traitement de la ligne
```

5) Organisation des fichiers sur disque

Les systèmes de stockage (disque dur, clé USB...) sont structurés par des systèmes de fichiers qui organisent²⁷ les séquences d'octets constituant les données des fichiers et permettent de les référencer avec des noms manipulables par un être humain ou par un programme.

a) Nommage

Pour les noms des répertoires et fichiers sur disque, il est très fortement déconseillé (carrément interdit sur certains systèmes d'exploitation) d'utiliser les caractères suivants : < > : " / \ | ? *

Dans les noms, il est aussi généralement déconseillé d'utiliser des caractères accentués, et il vaut mieux utiliser des caractères trait de soulignement (AltGr-8) _ à la place des espaces. Par exemple au lieu d'utiliser le nom "Mes Données" on utilisera le nom "Mes_Donnees".

b) Arborescence

Pour faciliter l'organisation des fichiers, ceux-ci sont regroupés dans une **arborescence à base de répertoires** pouvant contenir d'autres répertoires ou bien des fichiers. Certains répertoires sont réservés au système d'exploitation, d'autres à l'installation de logiciels, d'autres encore pour les données des utilisateurs, pour des fichiers temporaires...

La **racine de l'arborescence** se désigne par / sous Linux et MacOS X, et par C:\ sous Windows (le C: désigne le volume de stockage, Windows a une racine d'arborescence par volume).

c) Séparateur de noms dans l'arborescence

Sous Windows, le caractère de séparation entre deux niveaux dans l'arborescence est le caractère \.

Malheureusement, c'est aussi le caractère d'échappement dans les chaînes de caractère en Python ('\\n' pour retour à la ligne, '\\t' pour tabulation, etc). Ceci est cause de nombreuses erreurs et incompréhensions.

Il existe plusieurs solutions pour éviter ce problème :

- Échapper les caractères \ dans les chemins de fichiers en les doublant, "C:\Users\moi\Documents" devient "C:\\Users\\moi\\Documents".
- Utiliser des « raw string » ou chaînes brutes, dans lesquelles les \ ne sont pas interprétés, ceci se fait en précédant l'expression littérale de la chaîne par un r : r"C:\Users\moi\Documents".
- Utiliser comme séparateur le caractère / : "C:/Users/moi/Documents".

Sous **Linux** ou MacOS X, le caractère de séparation entre deux niveaux dans l'arborescence est le caractère /.

Le problème ci-dessus ne se pose donc pas (sauf si on veut insérer un \ dans un nom de fichier, mais ceci est très fortement déconseillé !).

d) Notion de répertoire courant

Lorsqu'on démarre un programme, celui-ci définit un répertoire courant, qui est le répertoire dans lequel seront par défaut cherchés les fichiers à lire ou enregistrés les fichiers à écrire pour lesquels on n'a spécifié que le nom (aucune indication de répertoire).

Par défaut, pour les logiciels scripts et ligne de commande (comme ceux que l'on écrit en Python), le répertoire courant est le répertoire dans lequel on est lors du lancement du programme, souvent le répertoire personnel de l'utilisateur. Les logiciels de bureautique définissent souvent le répertoire courant comme étant « Mes Documents ».

Il est possible de connaître et de modifier le répertoire courant :

```
>>> import os
>>> os.getcwd()
'/home/laurent/provisoire'
>>> os.chdir('/home/laurent/')
>>> os.getcwd()
'/home/laurent'
```

²⁷ - Les séquences d'octets des fichiers sont regroupées dans des séries de blocs de taille fixe (par 512 ou 1024 ou 2048 ou 4096 octets), pas forcément contigus sur le disque. Le système de fichiers se charge de donner aux programmes une vision séquentielle de ces données.

e) Chemins de fichiers

Le chemin d'un fichier (« path » en anglais) est la liste des répertoires qu'il faut traverser pour pouvoir accéder à ce fichier.

Si ce chemin commence par l'indication de racine de l'arborescence (/ sous Linux ou MacOSX, C:\ ou \ sous Windows), alors le chemin est un **chemin absolu** exprimé à partir de cette racine. On part de la racine et on traverse les répertoires intermédiaires pour arriver au fichier.

Si le chemin ne commence pas par une indication de racine de l'arborescence, mais directement par un nom (de répertoire ou bien le nom du fichier), alors il s'agit d'un **chemin relatif au répertoire courant**. On part du répertoire courant et on traverse les répertoires intermédiaires pour arriver au fichier.

Deux noms de répertoires spécifiques permettent de **naviguer entre les niveaux de répertoires** : `.` représente le répertoire courant (en le traversant on reste dans le même répertoire), et `..` représente le répertoire parent (en le traversant on remonte d'un niveau dans l'arborescence).

Python fournit une série d'**outils de manipulation de chemins**, dans les module `os` et `os.path` (ainsi que dans le récent module `pathlib`). Il est fortement conseillé d'utiliser ces outils pour manipuler les chemins de fichiers d'une façon qui soit portable sur les différents systèmes d'exploitation.

```
>>> import os.path
>>> os.path.split("/usr/local/bin/cargo")
('/usr/local/bin', 'cargo')
>>> os.path.dirname("/usr/local/bin/cargo")
'/usr/local/bin'
>>> os.path.splitext("monfichier.txt")
('monfichier', '.txt')
>>> os.path.join('/home', 'laurent', 'essai.txt')
'/home/laurent/essai.txt'
>>> os.path.join(os.getcwd(), 'essai.txt')
'/home/laurent/essai.txt'
>>> os.path.abspath('essai.txt')
'/home/laurent/essai.txt'
```

XI - Les exceptions

Le mécanisme d'exceptions est une façon de traiter les erreurs qui se produisent au cours de l'exécution d'un programme²⁸.

1) Capture des exceptions

L'idée est de séparer le bloc normal d'instructions (quand tout se passe bien) des blocs d'instructions qui traitent les erreurs. Un exemple :

```
try:
    # Lorsque tout se passe bien, on exécute uniquement ce bloc
    x = int(input("Valeur de x:"))
    y = int(input("Valeur de y:"))
    quot = x // y
    rest = x % y
    print("Quotient:", quot, "et reste:", rest)
except ValueError as e:
    # En cas d'erreur de conversion en entier, une exception ValueError est levée
    print("Saisie nombre entier invalide.", e)
except ZeroDivisionError:
    # En cas de division par zéro, une exception ZeroDivisionError est levée
    print("Division par zéro!")
```

En fonctionnement sans erreur, on n'exécute que le bloc **try**. Mais si une des instructions de ce bloc (ou une instruction d'une fonction appelée à partir de ce bloc) détecte une erreur et lève une exception, alors l'exécution passe immédiatement au premier bloc **except** prévu pour traiter ce genre d'exception, on dit que l'exception est « capturée » (un **except** qui ne spécifie pas de type d'exception capture toutes les exceptions).

Si aucun bloc **except** lié au **try** ne traite l'exception, elle remonte dans les fonctions qui ont conduit à appeler ce code, jusqu'à trouver un bloc **except** capable de traiter l'exception, ou jusqu'à terminer le programme si l'exception n'est pas capturée.

L'objet qui représente l'exception peut être récupéré dans une variable avec la notation :

```
except UneException as e :
```

Cela permet d'aller éventuellement chercher des compléments d'informations sur l'erreur dans cet objet.

2) Traitement des exceptions

Lorsqu'une exception est capturée (un bloc **except** activé suite à la détection d'une erreur), à la fin de l'exécution du traitement d'erreur il est possible :

- De continuer normalement l'exécution (après l'ensemble **try/except**) en ne faisant rien de spécial dans le bloc de traitement de l'exception, par exemple parce qu'on a corrigé l'erreur et on peut continuer le traitement à la suite.

Si on désire *recommencer le traitement qui a échoué*, alors le bloc **try/except** sera dans le corps d'une boucle **while** qui permette de refaire une tentative d'exécution.

- De faire remonter l'erreur à un plus haut niveau avec une instruction **raise** seule, par exemple parce qu'en l'état on ne peut pas continuer à exécuter la suite du code.

👉 Dans tous les cas il est important de ne pas passer les erreurs silencieusement, il faut en garder une trace quelque part (un affichage avec **print**, un enregistrement dans un fichier texte de « logs »).

Dans le traitement possible des erreurs, Python permet aussi de mettre en place :

- Un bloc **else** qui est exécuté lorsque le bloc **try** a pu complètement s'exécuter sans levée d'exception.
- Un bloc **finally** qui est exécuté dans tous les cas après le bloc **try** ou **try/else** ou après un **try/except** (si une exception a été levée). Ce bloc permet typiquement de libérer des ressources allouées (par exemple de s'assurer qu'on referme un fichier ouvert).

²⁸ - Une autre façon de procéder utilise des codes de retour et/ou une variable globale d'erreur, à vérifier après les appels aux fonctions pour savoir si une erreur s'est produite — le langage C par exemple travaille essentiellement de cette façon.

Pour schématiser, on a une structure de blocs consécutifs :

```
try:
    ► bloc d'instructions du traitement normal
except Exception1:
    ► bloc d'instructions pour le cas où l'exception1 se produit
except Exception2:
    ► bloc d'instructions pour le cas où l'exception2 se produit
except:
    ► bloc d'instructions pour le cas où n'importe quelle autre exception se produit
else:
    ► bloc d'instructions pour le cas où aucune exception n'est produite
finally:
    ► bloc d'instructions qui s'exécutera dans tous les cas
```

Exemple complet :

```
1. #!/usr/bin/python3
2. # -*- coding: UTF-8 -*-
3. """pour tester la gestion des exceptions"""
4. # fichier : ex_exceptions.py
5.
6. from math import sqrt
7.
8. # Première fonction
9. def diviser(a,b):
10.     quotient = a/b
11.     return quotient
12.
13. # Deuxième fonction
14. def diviser(a,b):
15.     try:
16.         quotient = a/b
17.     except :
18.         print("Une erreur s'est produite.")
19.         quotient = None
20.     else :
21.         print("La division s'est bien passée")
22.     finally :
23.         return quotient
24.
25. # Troisième fonction
26. def diviser(a,b):
27.     try:
28.         quotient = a/b
29.     except ZeroDivisionError :
30.         print("Il est interdit de diviser par 0.")
31.         quotient = None
32.     except TypeError :
33.         print("Il n'est pas possible de diviser ces types de données.")
34.         quotient = None
35.     except :
36.         print("Une erreur s'est produite")
37.         quotient = None
38.     finally :
39.         return quotient
40.
41. # Quatrième fonction
42. def diviser(a,b):
43.     try:
44.         quotient = a/sqrt(b)
45.         #if b == 1:
46.             #raise ValueError("diviseur égal à 1")
47.     except ZeroDivisionError :
48.         print("Il est interdit de diviser par 0.")
49.         quotient = None
50.     except TypeError :
51.         print("Il n'est pas possible de diviser ces types de données.")
52.         quotient = None
53.     except :
54.         print("Une erreur s'est produite")
55.         quotient = None
56.     finally :
57.         return quotient
58.
```



```

59. if __name__ == "__main__" :
60.     val = diviser(10,2)
61.     print("10/2 =",val, '\n')
62.
63.     val = diviser(10,0)
64.     #ZeroDivisionError: division by zero
65.     print("10/0 =",val, '\n')
66.
67.     val = diviser("bonjour", "bonsoir")
68.     #TypeError: unsupported operand type(s) for /: 'str' and 'str'
69.     print("bonjour/bonsoir =",val, '\n')
70.
71.     val = diviser(2,-3)
72.     print("2/(-3) =",val, '\n')
73.     #ValueError si on met a/sqrt(b) dans la fonction diviser()
74.
75.     val = diviser(35,1)
76.     print("35/1 =",val, '\n')
77.
78.     val = diviser(35,5)
79.     print("35/5 =",val)

```

3) Signalisation d'erreur : levée d'exception

Les bibliothèques Python et le langage génèrent automatiquement des exceptions lorsqu'ils détectent des erreurs. Il est possible de signaler les erreurs que nous détectons dans notre code en levant directement des exceptions standard Python :

```

v = int(input("Valeur de x entier (strictement positif):"))
if x <= 0 :
    raise ValueError("Valeur x saisie négative ou nulle")

```

Il est aussi possible de créer ses propres classes d'exception en dérivant soit la classe `Exception`, soit une de ses sous-classes²⁹. Cela permet de mettre en place des blocs de traitement d'exception dédiés.

Par exemple, la classe d'exception suivante :

```

class PasBienError(Exception):
    def __init__(self, x):
        super().__init__(self, x)

```

Sera utilisée de la façon suivante :

```

raise PasBienError("Ouh, c'est pas bien")

```

Et pourra être interceptée par :

```

try :
    ► ...
except PasBienError:
    ► ...

```

Si elle n'est pas interceptée, elle produira une indication d'erreur :

```

Traceback (most recent call last):
  File "<console>", line 1, in <module>
PasBienError: (PasBienError(...), "Ouh, c'est pas bien")

```

29 - Voir la hiérarchie des exceptions Python : <https://docs.python.org/3/library/exceptions.html>

XII - Programmation Orientée Objet

La programmation orientée objet (POO) consiste à regrouper les données qui sont liées entre elles en tant qu'**attributs** de structures qu'on appelle des **objets**, et de pouvoir associer à ces objets des fonctions de traitement qui leur sont spécifiques qu'on appelle des **méthodes**. Les méthodes sont appelées directement à partir de l'objet.

1) Utilisation des objets

Vous avez déjà utilisé des objets et des méthodes... par exemple `append()` et `pop()` pour les listes ainsi que `upper()` pour les chaînes, sont des méthodes :

```
>>> l1 = ["titi","tata"]
>>> l1.append("toto")
>>> l1
['titi', 'tata', 'toto']
>>> l1.pop()
'toto'
>>> l1
['titi', 'tata']
>>> machaine = "Un texte"
>>> machaine.upper()
'UN TEXTE'
```

Les listes et les chaînes de caractères sont des objets... car **en Python tout est objet**, même les nombres ont des méthodes et des attributs (l'accès aux attributs se fait comme l'accès aux méthodes, sans mettre de parenthèse) :

```
>>> n = 23
>>> n.numerator
23
>>> n.denominator
1
>>> v = 32.5
>>> v.as_integer_ratio()
(65, 2)
```

2) Création de familles d'objets... les classes

En Python la création de nouvelles familles d'objets passe par la création de **classes**³⁰ qui décrivent ces objets avec leurs attributs et leurs méthodes. Des méthodes particulières, dont le nom est entouré par deux `_` de part et d'autre, sont appelées de façon transparente par Python à des moments particuliers.

Un exemple, qui sera commenté après :

```
1. from math import sqrt
2.
3. class Point:
4.     def __init__(self, x, y, z):
5.         self.x = x
6.         self.y = y
7.         self.z = z
8.     def __str__(self):
9.         return "Pt({},{},{})".format(self.x, self.y, self.z)
10.    def est_origine(self):
11.        return (self.x == 0 and self.y == 0 and self.z == 0)
12.
13. class Segment:
14.     def __init__(self, ptdebut, ptfin):
15.         self.pt1 = ptdebut
16.         self.pt2 = ptfin
17.     def longueur(self):
18.         return sqrt((self.pt2.x - self.pt1.x) ** 2 +
19.                     (self.pt2.y - self.pt1.y) ** 2 +
20.                     (self.pt2.z - self.pt1.z) ** 2)
21.     def __str__(self):
22.         return "Seg({} à {})".format(self.pt1, self.pt2)
23.
```

30 - D'autres langages utilisent d'autres mécanismes, par exemple Javascript utilise les « prototypes ».

```

24. class Vecteur(Point):
25.     def __init__(self, x, y, z):
26.         super().__init__(x, y, z)
27.     def distance(self):
28.         return sqrt(self.x ** 2 + self.y ** 2 + self.z ** 2)
29.     def __str__(self):
30.         return "Vect({}, {}, {})".format(self.x, self.y, self.z)
31.     def __add__(self, autre_vect):
32.         nouveau = Vecteur(self.x + autre_vect.x,
33.                             self.y + autre_vect.y,
34.                             self.z + autre_vect.z)
35.         return nouveau
36.     def __mul__(self, coef):
37.         nouveau = Vecteur(self.x * coef,
38.                             self.y * coef,
39.                             self.z * coef)
40.         return nouveau
41.
42. p1 = Point(3,2,4)
43. p2 = Point(1,5,2)
44. p3 = Point(4,-3,8)
45. p4 = Point(0,0,0)
46. print("Points:", p1, p2, p3, p4)
47. print("Origine:", p1.est_origine(), p2.est_origine(),
48.         p3.est_origine(), p4.est_origine())
49.
50. s1 = Segment(p1, p2)
51. s2 = Segment(p1, p3)
52. print("Segments:", s1, s2)
53. print("Longueurs:", s1.longueur(), s2.longueur())
54.
55. v1 = Vecteur(5,-2,8)
56. v2 = Vecteur(8,2,-3)
57. v3 = v1 + v2
58. v4 = v1 * 3
59. v5 = v1 + Vecteur(-5,2,-8)
60. print("Vecteurs:", v1, v2, v3, v4)
61. print("Distances:", v1.distance(), v2.distance(),
62.         v3.distance(), v4.distance(),
63.         v5.distance())
64. print("Origine:", v1.est_origine(), v2.est_origine(),
65.         v3.est_origine(), v4.est_origine(),
66.         v5.est_origine())
67. """
68. Points: Pt(3,2,4) Pt(1,5,2) Pt(4,-3,8) Pt(0,0,0)
69. Origine: False False False True
70. Segments: Seg(Pt(3,2,4) à Pt(1,5,2)) Seg(Pt(3,2,4) à Pt(4,-3,8))
71. Longueurs: 4.123105625617661 6.48074069840786
72. Vecteurs: Vect(5, -2, 8) Vect(8, 2, -3) Vect(13, 0, 5) Vect(15, -6, 24) Vect(0, 0, 0)
73. Distances: 9.643650760992955 8.774964387392123 13.92838827718412 28.930952282978865 0.0
74. Origine: False False False False True
75. """

```

Cet exemple illustre plusieurs choses :

- La **création d'un nouvel objet à partir de sa classe** se fait en utilisant le nom de la classe avec des parenthèses, comme pour un appel de fonction.
- Une **méthode spéciale `__init__()`** est appelée automatiquement, avec les arguments donnés pour la création de l'objet.
- Le **premier paramètre `self` des méthodes** reçoit l'objet manipulé. Tous les accès aux attributs ou aux méthodes de l'objet passent par ce paramètre. Lors de l'appel d'une méthode par la notation `variable.methode()`, la variable placée avant le point est transmise de façon implicite comme argument pour le premier paramètre (`self`) de la méthode.
- Une **méthode spéciale `__str__()`** est appelée automatiquement lorsqu'il y a besoin d'une représentation textuelle de l'objet (par exemple la fonction `print` appelle cette méthode de façon transparente). Elle doit retourner une chaîne de caractères.
- Un objet peut être utilisé en tant qu'attribut pour *composer* d'autres objets, par exemple des `Point` sont utilisés pour créer des `Segment`.

- Il est possible de créer une **sous-classe** d'une classe existante, comme `Vecteur` est une sous-classe de `Point`. La sous-classe doit appeler la méthode de construction de sa *classe parents* (avec `super()`). Elle **hérite** de tous les *attributs* et *méthodes* de la classe parente (on peut accéder à `x`, `y`, `z` et appeler `est_origine()` sur les objets `Vecteur`). Elle peut définir ses propres méthodes, si elle définit une méthode de même nom qu'une méthode de la classe parente, elle masque la méthode de la classe parente (`__str__()` de `Vecteur` est appelé pour les vecteurs).
- Une **méthode spéciale** `__add__()` est appelée automatiquement lorsqu'un opérateur `+` est rencontré.
- Une **méthode spéciale** `__mul__()` est appelée automatiquement lorsqu'un opérateur `*` est rencontré.

Note : vous pouvez retrouver la **liste des méthodes spéciales** dans la documentation Python, ou encore sur l'Abrégé Dense Python 3.2 (<https://perso.limsi.fr/pointal/python:abrege>).

XIII - Le module matplotlib

Le module `matplotlib` permet de tracer facilement de nombreux types de courbes, d'y effectuer des réglages (couleurs, légendes, échelles...), et de les afficher ou bien de générer des fichiers image que l'on peut ensuite intégrer dans d'autres logiciels.

La documentation de matplotlib est visible sur le site internet matplotlib.org :

- http://matplotlib.org/users/pyplot_tutorial.html
- <http://matplotlib.org/gallery.html>
- <http://matplotlib.org/examples/>
- <http://matplotlib.org/contents.html>
- http://matplotlib.org/api/pyplot_api.html
- <http://matplotlib.org/users/artists.html>

Le module `matplotlib` est un *module tiers*, installé en plus des modules standard de Python. S'il est déjà installé, l'instruction suivante ne doit pas générer d'erreur d'import :

```
>>> import matplotlib
```

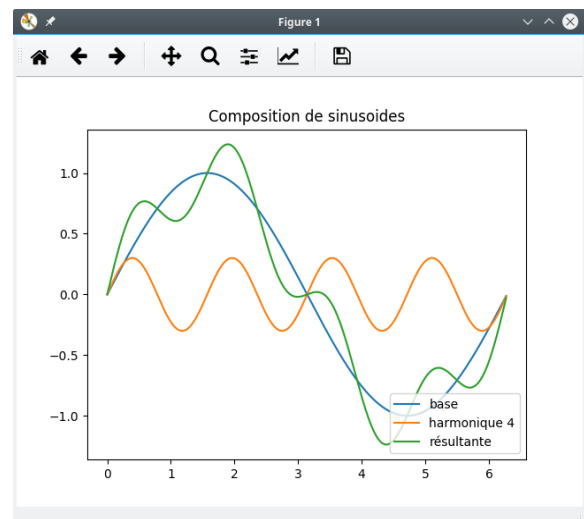
S'il n'est pas installé, vous pouvez l'installer en suivant les instructions données sur le site de matplotlib : <http://matplotlib.org/users/installing.html>

1) Utilisation

Un exemple pour tracer une série de courbes $x \mapsto \sin(x)$, en créant une liste `liste_x` contenant des valeurs de 0 à 6.28, et des listes `liste_y`, `liste_y2` et `liste_res` contenant les valeurs des sinus et de somme de sinus correspondants :

```
1. from matplotlib import pyplot as plt
2. from math import sin
3.
4. # On crée des listes de valeurs
5. liste_x = [x/100 for x in range(628)]
6. liste_y = [sin(x) for x in liste_x]
7. liste_y2 = [0.3*sin(4*x) for x in liste_x]
8. liste_res = [a+b for a,b in zip(liste_y, liste_y2)]
9.
10. # On trace...
11. fig, ax = plt.subplots()
12. line1, = ax.plot(liste_x, liste_y, label="base")
13. line2, = ax.plot(liste_x, liste_y2, label="harmonique 4")
14. line3, = ax.plot(liste_x, liste_res, label="résultante")
15. ax.legend(loc="lower right")
16. plt.title("Composition de sinusoides")
17. plt.show()
```

Ce module offre de nombreuses options, des types de graphiques différents, des possibilités de réglages pour les composants des graphiques, la capacité à placer différents graphiques côte à côte ou superposés... le plus simple est souvent de parcourir la galerie, de reprendre le code d'exemple, et de l'adapter à ses besoins.



XIV - Annexes

1) Webographie

- CORDEAU, Bob & POINTAL, Laurent, **Introduction à Python 3**
 - <https://perso.limsi.fr/pointal/python:courspython3>
 - Une introduction à Python 3 (document pdf)
 - Exercices corrigé Python 3 (document pdf)
- SWINNEN, Gérard, **Apprendre à programmer avec Python 3**
 - <http://inforef.be/swi/python.htm>
 - Téléchargeable (document pdf), et existe sous forme imprimée (Eyrolles)
- POINTAL, Laurent, **Cours 2012-2013** (présentations, fiches récapitulatives, scripts)
 - https://perso.limsi.fr/pointal/python:cours_prog (documents pdf, texte sources)
- NAROUN, Kamel, **Débuter avec Python au Lycée**
 - <http://python.lycee.free.fr/> (lecture en ligne)
- POINTAL, Laurent, **Mémento Python 3**
 - <https://perso.limsi.fr/pointal/python:memento> (document pdf)
- LE GOF, Vincent, **Apprenez à programmer en Python, Le Livre du Zéro**, 2011,
 - <http://www.siteduzero.com/informatique/tutoriels/apprenez-a-programmer-en-python>
- **Manuel de référence python (en anglais)**
 - <http://www.python.org/> (cliquer sur DOCUMENTATION)
- **Sites web interactifs** permettant d'apprendre à programmer
 - **CodinGame** (en) : <https://www.codingame.com/start>
 - **Codecademy** (en [/fr?]) : <https://www.codecademy.com/fr/learn/python>
- **Tutoriels sur developpez.com**
 - <https://python.developpez.com/>
 - <https://python.developpez.com/cours/>
- **Wikibook python**
 - http://fr.wikibooks.org/wiki/Programmation_Python

2) Bibliographie

- BALKANSKI, Cécile et MAYNARD, Hélène, **Algorithmique et C++**, *Supports de TD et de TP*, IUT d'Orsay, département Informatique
- CHESNEAU Myriam, **Cours d'informatique - Initiation au langage C**, IUT d'Annecy, Département Mesures Physiques, 2010-2011
- CHUN, W. J., **Au Coeur de Python**, CampuPress, 2007.
- DABANCOURT, Christophe, **Apprendre à programmer, Algorithmes et conception objet**, Eyrolles, 2008, 2e édition.
- LUTZ, Mark & BAILLY, Yves, **Python précis et concis**, O'Reilly, 2e édition, 2005.
- MARTELLI, Alex, **Python par l'exemple**, O'Reilly, 2006.
- SWINNEN, Gérard, **Apprendre à programmer avec Python 3**, Eyrolles, 2010,

- SUMMERFIELD, Mark, *Programming in Python 3*, Addison-Wesley, 2e édition, 2009.
- YOUNKER, Jeff, *Foundations of Agile Python Development*, Apress, 2008.
- ZIADÉ, Tarek, *Programmation Python. Conception et optimisation*, Eyrolles, 2e édition, 2009.
- ZIADÉ, Tarek, *Python : Petit guide à l'usage du développeur agile*, Dunod, 2007.

3) Codes hexadécimaux

Tableau de correspondance des valeurs hexadécimales :

Code Hexadécimal	Valeur Binaire	Valeur Décimale
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
a	1010	10
b	1011	11
c	1100	12
d	1101	13
e	1110	14
f	1111	15

4) Priorité des opérateurs

Classés du moins au plus prioritaire :

Opérateurs	Description
lambda	Expression lambda (fonction anonyme)
if – else	Expression conditionnelle
or	Ou logique booléen
and	Et logique booléen
not x	Non logique booléen
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparaisons, en incluant le test d'appartenance et le test d'identité
	Ou bit à bit
^	Ou exclusif bit à bit
&	Et bit à bit
<<, >>	Décalages de bits

Opérateurs	Description
<code>+</code> , <code>-</code>	Addition et soustraction
<code>*</code> , <code>@</code> , <code>/</code> , <code>//</code> , <code>%</code>	Multiplication, multiplication matricielle, division et division entière, reste de division entière (et formatage de chaînes de caractères)
<code>+x</code> , <code>-x</code> , <code>~x</code>	Positif, négatif, non bit à bit (inversion des bits)
<code>**</code>	Exponentielle
<code>await x</code>	Expression d'attente (Python 3 ≥ 3.5)
<code>x[index]</code> , <code>x[index:index]</code> , <code>x(arguments...)</code> , <code>x.attribute</code>	Indexation, tranchage, appel fonction, accès à un attribut
<code>(expressions...)</code> , <code>[expressions...]</code> , <code>{key: value...}</code> , <code>{expressions...}</code>	Définition de tuple, de liste, de dictionnaire, d'ensemble

5) Récapitulatif des opérateurs booléens

Le tableau ci-après récapitule les opérateurs de la logique booléenne, en donnant différentes notations que vous pouvez trouver suivant les domaines où ils sont utilisés. Dans le tableau, F pour False (Faux), T pour True (Vrai).

<i>variables</i>	<i>non</i>	<i>et</i>	<i>ou (inclusif)</i>	<i>ou exclusif</i>	<i>non et</i>	<i>non ou</i>	<i>implique</i>	<i>équivalent</i>
a b	not a	a and b	a or b	a xor b	a nand b	a nor b	a impl b	a equ b
<i>maths</i>	$\neg a$	$a \wedge b$	$a \vee b$	$a \oplus b$	$a \uparrow b$	$a \downarrow b$	$a \Rightarrow b$	$a \Leftrightarrow b$
<i>ingénierie</i>	\bar{a}	$a \cdot b^{31}$	$a + b$	$a \oplus b$	$\overline{a \cdot b}$	$\overline{a + b}$	$a \Rightarrow b$	$a \odot b$
F F	T	F	F	F	T	T	T	T
F T	T	F	T	T	T	F	T	F
T F	F	F	T	T	T	F	F	F
T T	F	T	T	F	F	F	T	T
équivalence (notation maths) :				$(\neg a \wedge b) \vee (\neg b \wedge a)$	$\neg a \vee \neg b$	$\neg a \wedge \neg b$	$\neg a \vee b$	$(a \wedge b) \vee (\neg a \wedge \neg b)$

6) Différences entre Python 2 et Python 3

Pour l'enseignement nous avons choisi Python 3, qui corrige certains défauts de Python 2 et est maintenant la version « active » de Python. Si vous avez des développements à faire, choisissez prioritairement Python 3. Toutefois, il existe encore de nombreux programmes écrits dans la version précédente de Python, et vous pouvez donc être amenés à devoir travailler avec Python 2, voici donc une liste rapide des différences entre les deux auxquelles il vous faut prêter attention :

31 - Parfois on omet l'opérateur \cdot (« $a \cdot b$ » est directement noté « ab »), entre autres lorsqu'on a des variables logiques qui sont toutes composées d'un seul caractère.

Les principales différences entre python2 et python3 sont :

Python 2	Python 3	Notes
print est une instruction, elle ne prend pas de parenthèses pour être appelée <pre>>>> print "x=", 12</pre>	print est une fonction, elle est appelée avec des paramètres entre parenthèse <pre>>>> print("x=", 12)</pre>	from __future__ import print_function permet à Python 2 de définir print comme une fonction
la division / effectue une division entière <pre>>>> 1/2 0 >>> 20/3 6</pre>	la division / effectue une division décimale <pre>>>> 1/2 0.5 >>> 20/3 6.666666666666667</pre>	from __future__ import division permet à Python 2 d'utiliser / comme diviseur décimal
la fonction input() évalue la chaîne saisie (comme le fait le Shell Python) et renvoie une valeur avec le type correspondant la fonction raw_input() retourne la saisie sous forme de chaîne de caractères sans l'évaluer	la fonction input() retourne systématiquement la saisie sous forme de chaîne de caractères sans l'évaluer	
le type str stocke les chaînes sous une forme de séquence d'octets sans information sur leur encodage. il existe un type unicode (chaînes préfixées par u : u"Une chaîne").	le type str stocke les chaînes dans leur représentation Unicode, le type bytes stocke des séquence d'octets	from __future__ import unicode_literals permet à Python 2 de coder les chaînes littérales en tant que chaînes Unicode sans avoir à spécifier le u""
range() produit une liste, il existe xrange() qui retourne un générateur	range() est un générateur, si on veut une liste il faut la construire explicitement : list(range())	

Mais il y en a d'autres. Le module `six`³² fournit un support pour faciliter l'écriture de code qui soit compatible Python 2 et Python 3.

Plus de détails sur (en) http://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html

32 - Voir sur <https://pypi.python.org/pypi/six>