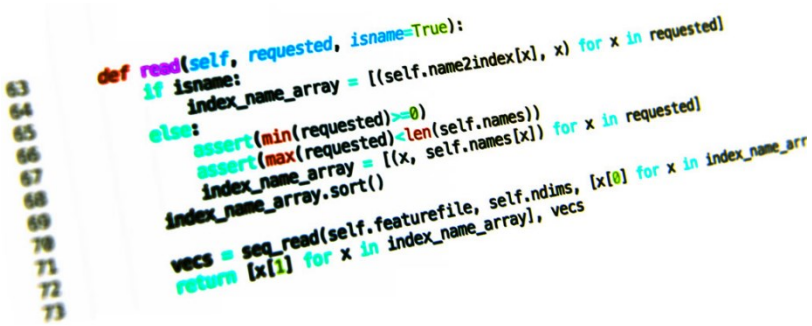
Premier programme et premiers pas en Python



Compétences visées

- Acquérir des méthodes de programmation
- Comprendre un jeu d'instructions et d'expressions
- Spécifier les données attendues en entrée, et fournies (retournées) en sortie
- Structurer la programmation en sous-problèmes
- Utiliser des modules, des bibliothèques

Table des matières

1. Structure de l'interface 2
1.1. Préambule syntaxique et exemple support 2
1.2. La console 3
1.3. L'éditeur 4
2. Structure de la programmation 6
2.1. Création du fichier vierge 6
2.2. En-tête et imports 6
2.3. Variables globales, portée de variable et fonctions 8
2.4. Programme principal 10
2.5. Post-traitement des résultats 10
3. Exercice de programmation 11

Les extraits de code seront encadrés sous la forme suivante, et les lignes numérotées :

```
1.      c = a + b  # première instruction
```

L'affichage résultant sur la console sera encadré en pointillé, outre la présence des « >>> » :

```
1.>>> somme(1,1)
```

1. Structure de l'interface

1.1. Préambule syntaxique et exemple support

Avant d'exécuter un programme, il faut commencer par l'enregistrer. Supposons défini le fichier toto.py contenant les lignes suivantes :

```
1.  # -*- coding: utf-8 -*-
2.  #!/usr/bin/env python
3.
4.  def somme(a,b) :
5.      c = a + b  # première instruction
6.      print("a + b =",c) # deuxième instruction
7.
8.  def sommeUser() :
9.      a = eval(input("valeur de a ?")) # eval convertit input en entier
10.     b = eval(input("valeur de a ?"))
11.     c = a + b
12.     print("a + b =",c)
```

Code 1. Fichier toto.py

Les ligne 1 et 2 sont un préambule obligatoire à tout programme python (voir section « En-tête et imports »).

La ligne 4 permet de définir une fonction, appelée somme, et dont le fonctionnement est conditionné par l'utilisation de deux variables « a » et « b » qui devront être spécifiée par l'utilisateur lors de l'appel de la fonction.

Pour les deux lignes d'instructions 5 et 6, une tabulation permet d'indiquer l'appartenance à la fonction somme. La fin de fonction est définie par la première rupture d'indentation (ligne 6 pour la fonction somme ci-dessus). Ce programme sera exécuté (F5). Il est alors possible d'utiliser les fonctions dans la console :

```
1.>>>
2.>>>somme(1,1)
3.a + b = 2
4.>>>sommeUser()
5.valeur de a ? 1
6.valeur de b ? 1
7.a + b = 2
```

Les lignes 5 et 6 ont provoqué une interaction avec l'utilisateur, qui a répondu à l'instruction « input » en donnant les valeurs de « a » et « b » (ici 1 et 1). Si à présent on rajoute, à la fin du fichier toto.py, les lignes :

```
13. ...
14.     print("a + b =",c) # deuxième instruction
15.
16. somme(1,1)
17. sommeUser()
```

L'exécution du fichier toto.py conduira directement à :

```
1.>>>
2.a + b = 2
3.valeur de a ? 1
4.valeur de b ? 1
5.a + b = 1
```

1.2. La console

Il est tout à fait possible d'utiliser python en lignes de commande (dans le terminal sous Linux, ou dans l'invite de commande Windows). On s'intéresse ici à l'interface accessible depuis la console IDLE.

❑ Présentation

Lorsque vous saisissez des commandes ou des calculs à la suite d'une invite de commande python dans IDLE ('>>>'), puis que vous pressez la touche « Entrée » de votre clavier, vos données sont envoyées à l'interpréteur Python pour y être exécutées.

Le résultat de votre requête s'affiche à la ligne suivante, toujours dans la console, puis de nouveau une invite de commande ('>>>') apparaît pour attendre vos instructions. Ces commandes rendent la console interactive.

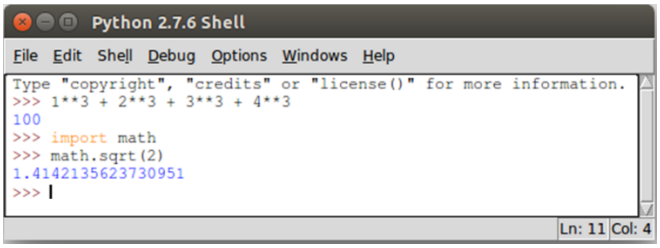


Fig. 1. Structure de la console IDLE

❑ Menu File

On y retrouve les options du menu fichier classique :

- **New File** (CTRL+N) : crée un nouveau fichier vierge dans une fenêtre Script ;
- **Open** (CTRL+O): ouvre un fichier existant à partir d'un emplacement donné ;
- **Save** (**As**) (CTRL+S et CTRL+SHIFT+S) : enregistre le contenu texte de la fenêtre actuelle (Shell ou Script) sous un nouveau chemin de fichier (emplacement et/ou nom).

L'option suivante est plus spécifique à la programmation :

- **Open module** (ALT+M) : ouvre une bibliothèque de fonction (voir la section « import » plus loin) pour présenter le code sous-jacent.

❑ Menu Edit

On y retrouve les options du menu éditer classique :

- **Undo** (CTRL+Z) : annule la dernière modification effectuée dans la fenêtre ;
- **Cut** (CTRL+X) : copie la sélection dans le presse-papiers puis efface la sélection ;
- **Copy** (CTRL+C) : copie la sélection dans le presse-papiers système ;
- **Paste** (CTRL+V) : insère le dernier contenu en date du presse-papiers ;
- **Find** (CTL+F) : ouvre une boîte de dialogue permettant de rechercher une expression dans la fenêtre concernée avec diverses options de recherche ;
- **Replace** (CTRL+H) : ouvre une boîte de dialogue Chercher / Remplacer.

On pourra également s'intéresser aux options de menu suivantes.

- **Show completions** (CTRL+SPACE) – Figure 1a : Affiche la liste des attributs et symboles disponibles parmi une liste déroulante (variables globales, fonctions, symboles Python standards (Error, fonctions internes), constantes du langage (True, None, etc.)) ;
- **Show surrounding parents** (Ctrl+0) – Figure 1b : Met en surbrillance le texte contenu entre les deux parenthèses englobantes les plus proches du curseur clignotant.

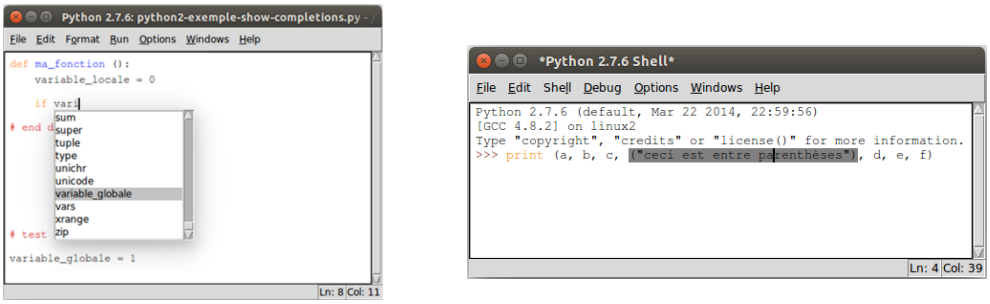


Fig. 2. Capture d'écran de l'interface ((a) Show completions, (b) Show surrounding parents)

❑ Menu Help

On y retrouve :

- **IDLE Help** : affiche une fenêtre d'aide sommaire sur l'utilisation d'IDLE (en anglais) ;
- **Python Docs** :
  - affiche la documentation Python locale à la machine, si celle-ci est installée ;
  - à défaut, lance le navigateur internet par défaut avec l'adresse « docs.python.org » correspondant à la version Python utilisée par le logiciel pour accéder à la documentation officielle Python.

1.3. L'éditeur

Les fichiers python peuvent parfaitement être créés avec un simple bloc-notes. On lui préfère généralement les éditeurs dédiés qui apportent une coloration syntaxique aux mots clés. Toute part des choses faites, la fonction de l'éditeur se borne à faire du traitement de texte.

❑ Présentation

Tout comme la console d'IDLE, la fenêtre Script se compose d'un menu, d'une zone de texte éditable et d'une barre clignotante indiquant la position du curseur.

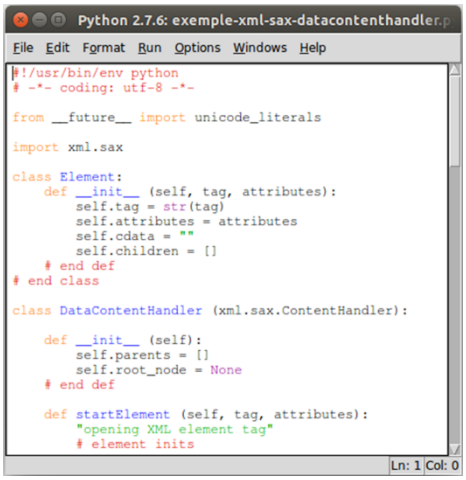


Fig. 3. Structure de la fenêtre d'édition

Les mêmes menus « File », « Edit » et « Help » que ceux définis précédemment pour la console sont disponibles dans l'éditeur IDLE.

Numéro de ligne

Les erreurs python ont la syntaxe suivante :

```
6.Traceback (most recent call last):
7.  File "<stdin>", line 1, in <module>
8.NameError: name 'spam' is not defined
```

Pour coder efficacement, il est donc indispensable de pouvoir se rendre facilement à une ligne donnée. Deux options s'offrent à nous :

- **Edit – Go to line** (ALT+G): ouvre une boîte de dialogue demandant un numéro de ligne puis place le curseur clignotant à cette ligne, si elle existe.
- **Option – Configure IDLE** : Onglet Shell/Ed : "Show line number in new window" : permet d'afficher les numéros de ligne lors de l'ouverture d'une prochaine fenêtre d'édition

Menu Format

En plus des menus issus de la console, un nouveau menu « Format » est disponible. Il est principalement utilisé pour (des)indenter et (de)commenter :

- **Indent region** (CTRL+]) : déplace le bloc de texte sélectionné d'un cran d'indentation vers la droite (quatre espaces blanches par défaut) ;
- **Dedent region** (CTRL+]) : déplace le bloc de texte sélectionné d'un cran de désindentation vers la gauche (quatre espaces blanches par défaut) ;

- **Comment out region** (CTRL+3) : insère les caractères de commentaire '#' en début de ligne(s) sélectionnée(s) ;
- **Uncomment region** (CTRL+4) : retire le(s) caractère(s) de commentaire '#' préalablement placé(s) en début de ligne(s) sélectionnée(s) ;

- **Tabify region** (CTRL+5) : remplace dans les lignes sélectionnées les espaces blanches d'indentation par des caractères de tabulation ('\t') équivalant à huit espaces blanches fixes par tabulation (non modifiable) ;
- **Untabify region** (CTRL+6) : remplace les caractères de tabulation ('\t') par le nombre d'espaces blanches défini dans la configuration d'indentation (quatre espaces blanches par défaut) ;

2. Structure de la programmation

2.1. Création du fichier vierge

Pour créer un fichier de programmation, on pourra s'inspirer de la structure ci-dessous :

```
1. # en-tête
2.
3. # définition des imports
4.
5. # définition des variables globales / des données
6.
7. # définition des fonctions
8.
9. # Programme principal
10.
11. # post-traitement
12.
```

Code 2. Organisation et contenu standard d'un programme

Note : On rappelle que ce qui est écrit après « # » n'est pas interprété lors de la compilation

2.2. En-tête et imports

❑ En-tête Linux

La séquence de caractères shebang est une séquence de caractères spéciale dans un fichier de script noté « #! ». Il indique au système d'exploitation que le fichier spécifié est un script informatique et non un fichier simple.

Il spécifie le programme qui doit être appelé pour exécuter l'intégralité du fichier de script (ici python). La séquence de caractères shebang est toujours utilisée dans la première ligne et spécifie l'endroit où peut être trouvé l'interpréteur.

```
1. #!/usr/bin/env python
```

Si python est installé à cet endroit (en principe).

❑ En-tête Windows

Windows utilise des shebang virtuels, et gère différemment les liens de telle sorte que ce lien est inutile. Le PATH utilisé par Windows est la variable système qui permet de localiser les fichiers exécutables indispensables. Il spécifiera où trouver Python sous Windows lors de l'exécution.

❑ En-tête commune

Les problèmes d'encoding viennent du fait que l'informatique a été développé en anglais et que dans cette langue il n'y a pas d'accents. A l'inverse, l'espagnol intègre des accents que nous ne possédons pas. Les encodages utilisés ne sont donc pas compatibles entre eux et cela créé des erreurs. Il est nécessaire de préciser l'encodage des caractères en en-tête du fichier. On se limitera à nommer l'UTF-8, qui présente l'avantage d'être universel. Il a pour objectif de réunir les caractères utilisés par toutes les langues. Il n'y aurait plus de problèmes de communication si tous les programmes étaient encodés en UTF-8, mais la mise à jour peine à se faire. La spécification de l'encodage se fera indifféremment avec une des deux lignes ci-dessous :

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

```
2. # coding: utf-8
```

Les imports

Afin d'utiliser des fonctions prédéfinies (comme par exemple les fonctions trigonométriques) sans les coder soi-même, il est possible d'importer des bibliothèques de fonctions.

Par exemple, on crée un fichier test.py, contenant les en-têtes définis précédemment. Dans un second fichier nommé puissance.py (le fichier puissance.py doit être dans le même répertoire que le programme principal (ou bien se trouver dans le path de Python)), contenant la définition de fonction suivante :

```
1.  # fonction racine carrée
2.  def rac(a) :
3.        return a**0.5
4.
```

Code 3. Définition d'une bibliothèque

On peut alors appeler la bibliothèque dans l'interpréteur, et l'utiliser directement comme fonction :

```
1.>>> import puissance
2.>>> puissance.rac(4)
3.2
4.>>>
```

Ou même dans notre fichier courant, test.py :

```
1.  #!/usr/bin/env python
2.  # -*- coding: utf-8 -*-
3.
4.  import puissance
5.
6.  a = 4
7.  print(puissance.rac(a))
8.
```

Code 4. Utilisation de la bibliothèque

Les bibliothèques natives de Python les plus utilisées en classe préparatoire sont : numpy, matplotlib et math. Numpy sera utilisé pour sa gestion des arrays (tableaux, vecteurs, …) ; matplotlib permettra de réaliser l'affichage des résultats dans le post-traitement des données ; enfin le module math est essentiel pour obtenir les fonctions d'algèbre et de calcul numérique.

Les syntaxes suivantes seront autorisées pour importer des fonctions d'une bibliothèque, voir une bibliothèque entière :

- import E : importe tout les fonctions de la bibliothèque E. L'accès aux fonctions de E se fait par la commande « E.nomFonction() »
- import geometry as D : donne l'alias D au module geometry. L'accès aux fonctions de geometry se fait par la commande « D.nomFonction() » ce qui permet de synthétiser les notations. Par exemple :

```
1.  #!/usr/bin/env python
2.  # -*- coding: utf-8 -*-
3.
4.  import puissance as pp
5.
6.  a = 4
7.  print(pp.rac(a))
```

Code 5. Utilisation de la bibliothèque avec l'utilisation d'alias

Exercice : L'utilisation de ALT+M sur IDLE permet d'ouvrir un module situé dans le path python. Importer le module math et aller voir à quoi ressemble sa structure !

Vous pouvez également importer un module du package (ou une fonction de la bibliothèque)

- from A import f1, f2 : importe les fonctions f1 et f2 de A. Il n'est plus nécessaire de préciser la bibliothèque. L'accès à la fonction est plus synthétique (on écrira simplement « f1() ».
- from G import * : importe toutes les fonctions de la bibliothèque A. Cette notation est fortement déconseillée car elle ne permet pas de savoir précisément ce qui a été importé. Elle est génératrice d'erreur pour les programmes complexes.

Exercice : TP2

Question 1 – Activer les numéros de ligne dans l'éditeur IDLE.

Question 2 – Créer un fichier « TP2.py », qui contiendra le programme principal et un fichier « test.py », qui contiendra la bibliothèque code 1 (voir page 1). Organiser votre bureau pour accéder facilement aux deux fenêtres d'édition et à la console.

Question 3 – Dans le fichier « TP2.py », copier-coller le code 2 pour établir la structure de votre programmation. Compléter l'en-tête avec l'encodage et le shebang.

Question 4 – Importer depuis le module « math », la fonction « cos »

Question 5 – Importer votre bibliothèque personnelle sous l'alias « t0 »

2.3. Variables globales, portée de variable et fonctions

Variables globales et données du problème

Par défaut, si une fonction Python utilise une variable dans le corps de la fonction, celle-ci est locale à la fonction. Pour vous en convaincre, considérons les lignes de code suivantes :

```
1.  #!/usr/bin/env python
2.  # -*- coding: utf-8 -*-
3.
4.  # définition des imports
5.
6.  import puissance as pp
7.
8.  # définition des variables globales
9.
10. a = 1
11.
12. # définition des fonctions
13. # ----------------------
14. def fonc() :
15.      a = 0
16.      for i in range(10)
17.           a+=1
18.      print("valeur a fonction ", a)
19.
20.
21. # Programme principal
22. # --------------------
23.
24. print("avant valeur de a ",a)
25. fonc()
26. print("apres valeur de a ",a)
```

Code 6. Illustration des portées des variables

Les résultats dans la console seront alors :

```
1.>>>
2.avant valeur de a 1
3.valeur a fonction 10
4.apres valeur de a 1
```

Pour que dans cet exemple, la fonction utilise le « a » défini ligne 10, on précisera que la variable a, utilisée dans la fonction, est celle fournie ligne 10 :

```
12. # définition des fonctions
13. # ----------------------
14. def fonc() :
15.       global a # on precise que a est donné comme variable globale
16.       a = 0
17.       for i in range(10)
18.             a+=1
19.       print("valeur a fonction ", a)
```

Les résultats dans la console seront les suivants :

```
5.>>>
6.avant valeur de a 1
7.valeur a fonction 11
8.apres valeur de a 11
```

Note : Des syntaxes dans l'appel des fonctions et des données retournées permettent généralement de s'affranchir de l'utilisation des variables globales. Les données du problème sont alors fournies dans l'argument des fonctions (voir cours à venir sur les fonctions).

Ainsi, notez bien que l'utilisation de variables globales
est à bannir de toute bonne pratique de programmation.

Parfois on veut faire vite et on crée une variable globale visible partout dans le programme (donc dans toutes les fonctions), car « c'est plus simple *et plus rapide ». C'est un très mauvais calcul, ne serait-ce que parce que vos fonctions ne seront pas réutilisables dans un autre contexte ! Arriverez-vous à vous relire dans six mois ? Quelqu'un d'autre pourrait-il comprendre votre programme ? Il existe de nombreuses autres raisons que nous ne développerons pas (complexité, occupation mémoire, …). Bref, si on peut faire autrement, on ne fait pas.*

❑ Définition des fonctions intermédiaires

En programmation, les fonctions sont très utiles pour réaliser plusieurs fois la même opération au sein d'un programme. Elles rendent également le code plus lisible et plus clair en le fractionnant en blocs logiques plus élémentaires.

Un programme réel est donc souvent sous-découpé en fonctions élémentaires. Par exemple, pour programmer le déplacement séquentiel d'un robot (par étapes), on définira les fonctions « avancer », « reculer » et « pivoter ».

Une fonction effectue une tâche. Pour cela, elle reçoit éventuellement des arguments et renvoie éventuellement quelque chose.

L'algorithme utilisé au sein de la fonction n'intéresse pas directement l'utilisateur. Par exemple, il est inutile de savoir comment la fonction math.cos() calcule un cosinus. On a juste besoin de savoir qu'il faut lui passer en argument un angle en radian et qu'elle renvoie un réel, valeur du cosinus de cet angle. Ce qui se passe à l'intérieur de la fonction ne regarde que le programmeur.

Exercice : TP2

Question 6 – Définir, dans la section adéquates deux données « a » et « b » valant respectivement π et 180.

La fonction « math.cos(angle) » du module « math » renvoie le cosinus de la variable angle exprimé en radian.

Question 7 – Définir une fonction « cosDeg » dans votre fichier « TP2.py », prenant en argument une valeur d'angle en degré, et renvoyant son cosinus en utilisant la fonction « cos » du module « math ».

Note : Toutes les fonctions seront définies au même endroit, dotées des commentaires adéquates pour en faciliter la lecture, en utilisant la structure présentée Code 2.

Chaque fonction effectue en général une tâche unique et précise. Si cela se complique, il est plus judicieux d'écrire plusieurs fonctions (qui peuvent éventuellement s'appeler les unes les autres, voire même s'appeler elles-mêmes (on parle alors de fonction récursive). Cette modularité améliore la qualité générale et la lisibilité du code.

2.4. Programme principal

On utilise le terme programme principal (main en anglais) pour désigner l'endroit depuis lequel on appelle les fonctions.

Le programme principal désigne le code qui est exécuté lorsqu'on lance le script Python, c'est-à-dire toute la suite d'instructions en dehors des fonctions. En général, dans un script Python, on écrit d'abord les fonctions puis le programme principal (comme dans la structure code 2).

L'appel des fonctions permet d'obtenir des résultats, qu'on affichera et post-traitera dans la section finale dédiée.

Exercice : TP2

Question 8 – Définir, dans la section adéquate, les instructions permettant de vérifier que les deux valeurs de cos (module math) de π et de cosDeg (définie dans les fonctions) de 180 sont bien égales à 10^{-2} près. Le programme renverra « True » si les deux valeurs sont corrélées.

2.5. Post-traitement des résultats

Le post-traitement induit souvent l'utilisation des fonctions intermédiaires complémentaires (graphiques, de traitement de tableaux, …). Idéalement, celles-ci seront définies dans une bibliothèque personnelle dédiée, pour ne pas surcharger la lecture du programme principal.

Les fonctions de post-traitement et les listes seront introduites plus tard dans l'année. A titre d'exemple, nous vous proposons d'importer le module pyplot de matplotlib en rajoutant en en-tête de fichier :

```
import matplotlib.pyplot as plp
```

Puis de copier/coller la portion de code ci-dessous dans votre programme principal :

```
1.  # post-traitement
2.
3.  a2 = [i for i in range(360)]
4.  b2 = [pi/180*i for i in range(360)]
5.
6.  liste0, liste1, liste2 = [],[],[]
7.  for i in range(360):
8.      liste0.append(i)
9.      liste1.append(cos(b2[i]))
10.     liste2.append(cosDeg(a2[i]))
11.
12. plt.figure(1)              # cree une figure numérotée 1
13. plt.plot(liste0,liste1)    # trace sur la figure
14. plt.plot(liste0,liste2)    # trace sur la figure
15. plt.show()                 # realise l'affichage des deux courbes
```

Code 7. Exemple de post-traitement des résultats

Note : La boucle for (introduite ultérieurement) est une structure permettant de réaliser le jeu d'instruction indenté (lignes 8 à 10) pour i variant de 0 à 359 (donc 360 fois). Pour mieux comprendre son fonctionnement, on pourra tester le jeu d'instruction page suivante dans la structure du programme courant.

```
1. sum0 = 0
2. a0 = input("nombre d'entiers à sommer ?")
3. for i in range(a0):
4.     print(i)
5.     sum0 = sum0 + i
6. print("somme des ", a0, " premiers entiers : ", sum0)
```

Code 8. Exemple de boucle for (afin d'effectuer la somme des premiers entiers)

Exercice : TP2

Question 9 – Observer les modifications apportées à l'affichage lorsque vous remplacez la ligne 13 par plt.plot(liste0,liste1, linewidth=5.0, color='r', linestyle='--', label='courbe1').

Question 10 – Ajouter l'instruction ligne 15 « plt.legend() » et observer le résultat.

3. Exercice de programmation

A partir du programme réalisé pendant la séance, on cherche à vérifier l'approximation suivante pour le cosinus au voisinage de zéro (on parle de développement limité d'ordre 6 de la fonction cosinus au voisinage de 0) :

$$\cos(\alpha) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \vartheta(x^6)$$

On cherche à vérifier la valeur pour laquelle l'approximation d'ordre 6 est vérifiée à 10% près.

Question 1 – Définir dans le fichier « test.py » une fonction facto, prenant en argument un entier et retournant la factorielle de cet entier. On s'inspirera du code 8.

Question 2 – Importer cette fonction dans le fichier « TP2.py ».

Question 3 – Définir alors dans le fichier « test.py » une fonction cosDVL1, prenant en argument un angle en radian et donnant la valeur du développement limité d'ordre 6 du cosinus.

Question 2 – Ajouter son affichage à la section post-traitement

Question 3 – En vous inspirant du code 9 ci-dessous, déterminer la première valeur pour laquelle l'approximation réalisée est faite avec une erreur supérieure à 10%. La valeur sera affichée dans la section post-traitement sous la forme :

```
1.>>>
2.Approx ordre 1 valable jusqu'a ... rad
3.Approx ordre 2 valable jusqu'a ... rad
```

```
1. sauv = 0
2. for i in range(100):
3.     a = pi/1000*i                    # on travaille sur 0, pi/10
4.     b = (cosDVL1(a)-cos(a))/cos(a)   # calcul d'erreur relative
5.     if(b>0.01) :                     # si tolérance n'est plus vérifiée
6.           sauv = a                   # sauvegarde la valeur courante de a
7.           break                      # brise boucle for avant l'heure
```

Code 9. Exemple de boucle for (afin d'effectuer la somme des premiers entiers)