Master Sciences, Technologies, Santé
Mention Mathématiques, spécialité Enseignement des mathématiques

MHT063

Algorithmique et graphes, thèmes du second degré

PETIT MEMENTO PYTHON

Éric SOPENA
Eric.Sopena@labri.fr

SOMMAIRE

Chapitre 1. Introduction 2

Chapitre 2. Python pour la classe de seconde 4

- 2.1. Éléments du langage 4
- 2.2. Types de données élémentaires 4
- 2.3. Affectation et opérations d'entrée-sortie 5
- 2.4. Structures de contrôle 6
- 2.5. Quelques exemples de scripts Python 7
- 2.6. Traduction d'algorithmes en Python – Tableau de synthèse 7
- 2.7. Dessiner en Python 8

Chapitre 3. Pour aller (un petit peu) plus loin… 11

Chapitre 1. Introduction

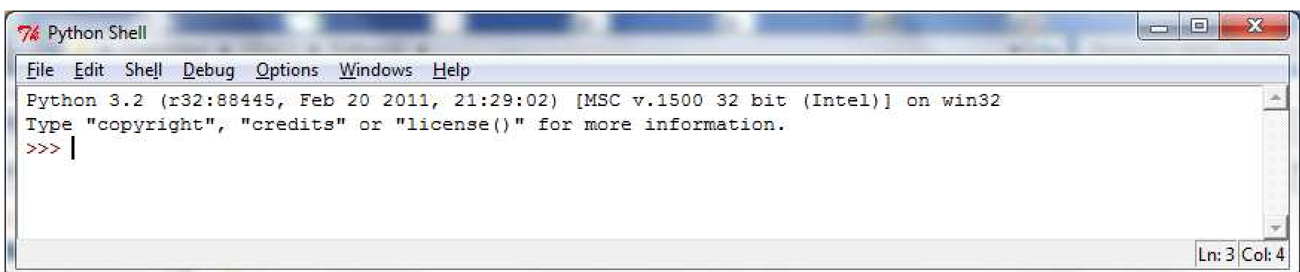
Le langage Python est né dans les années 1990, au CWI Amsterdam, développé par Guido van Rossum. Il a été nommé ainsi par référence à l'émission de la BBC « Monty Python's Flying Circus », et de nombreuses références aux dialogues des Monty Python parsèment les documentations officielles…

Python est un langage libre et gratuit, facile d'accès (il possède une syntaxe très simple), puissant, et est utilisé comme langage d'apprentissage par de nombreuses universités. Dans le cadre de l'initiation au lycée, seule une partie restreinte des possibilités de ce langage sera exploitée (en particulier, tous les aspects liés à la programmation par objets se situent hors du cadre de cet enseignement).

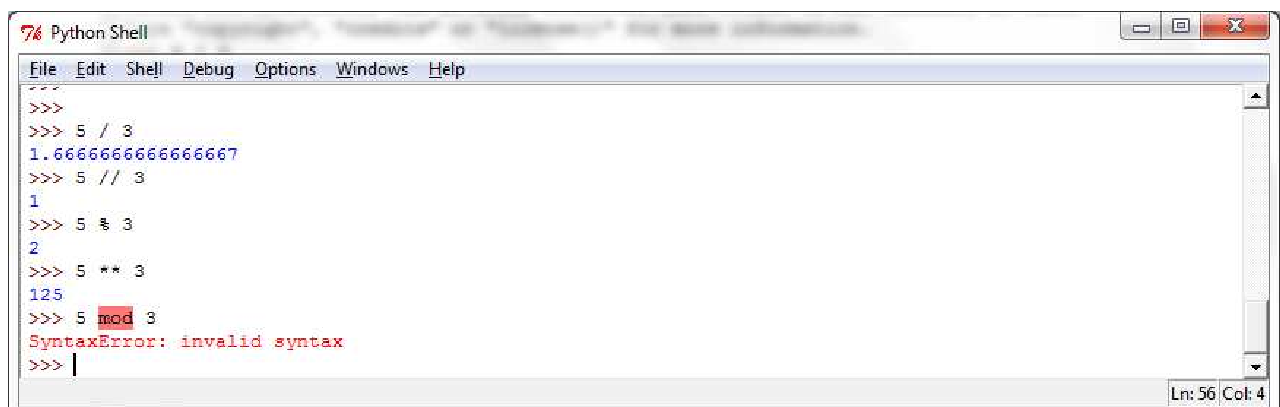
Le langage Python peut être utilisé en mode *interprété* (chaque ligne du code source est analysée et traduite au fur et à mesure en instructions directement exécutées) ou en mode *mixte* (le code source est compilé et traduit en *bytecode* qui est interprété par la *machine virtuelle* Python).

Le site officiel du langage Python est http://www.python.org/. On peut y télécharger gratuitement la dernière version du logiciel[1], pour différentes plateformes (Windows, Mac, Linux), ainsi que de nombreuses documentations. Notons que les versions 3.x constituent une réelle rupture avec les versions précédentes (2.x) qui ne sont plus maintenues. Ce document fait donc référence à la série des versions 3.x.

L'installation de Python ne pose aucun problème particulier. L'environnement de développement IDLE est fourni (<répertoire_Python>\Lib\idlelib\idle.bat) qui suffit amplement à l'utilisation du langage. Le lancement de IDLE ouvre la fenêtre suivante :

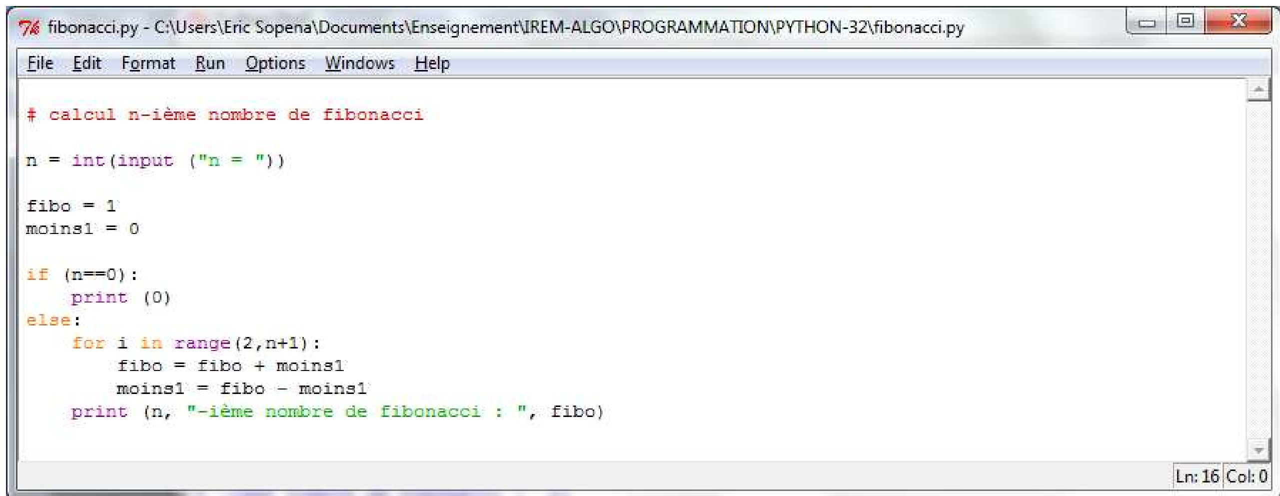


Il est alors possible d'utiliser directement Python en mode interprété (dans ce cas, Python lit, évalue et affiche la valeur) :

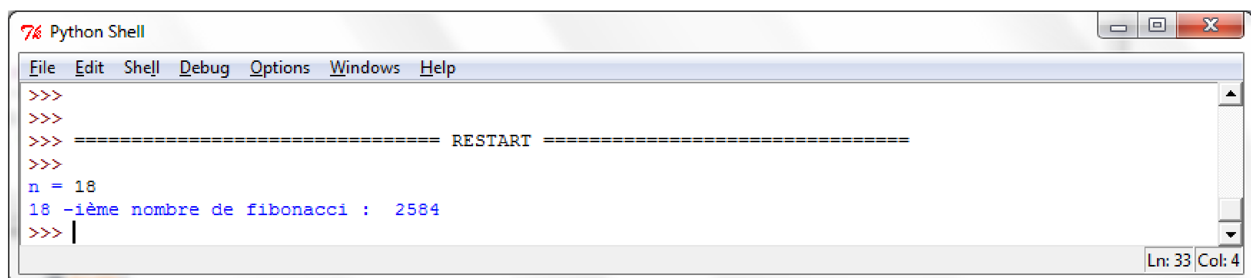


[1] À la date d'édition de ce document, il s'agit de la version 3.2.

ou d'éditer un *script* Python (menu File/New Window) :



Les scripts Python sont sauvegardés avec l'extension « .py ». On peut lancer l'exécution d'un script ouvert dans IDLE à l'aide de la touche F5 (ou par le menu Run/Run Module) :



Parmi les caractéristiques du langage Python, on notera en particulier les suivantes, qui diffèrent de celles que nous avons adoptées pour notre « langage algorithmique » :

- il n'est pas nécessaire de déclarer les variables (attention, cela peut être source de problème en phase d'initiation : une variable mal orthographiée sera considérée comme une nouvelle variable !…),
- les séquences d'instructions (ou blocs) sont définies en fonction de *l'indentation* (décalage en début de ligne) et non à l'aide de délimiteurs de type début-fin.

Ressources documentaires en ligne :

- www.python.org : site officiel de Python
- www.inforef.be/swi/python.htm : ressources didactiques (Gérard Swinnen), dont en particulier le livre *Apprendre à programmer avec Python 3* en téléchargement libre.
- http://hebergement.u-psud.fr/iut-orsay/Pedagogie/MPHY/Python/courspython3.pdf : cours de Robert Cordeau, *Introduction à Python 3*.

Chapitre 2. Python pour la classe de seconde

2.1. Éléments du langage

Un identificateur Python est une suite non vide de caractères, de longueur quelconque, à l'exception des *mots réservés* du langage : `and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, while, with, yield`. Notons que Python fait la distinction entre majuscules et minuscules. Ainsi, `nombre` et `Nombre` correspondent à des identificateurs différents. Une excellente habitude consiste à nommer les constantes en `MAJUSCULES` et les variables en `minuscules`.

Un commentaire Python commence par le caractère # et s'étend jusqu'à la fin de la ligne :

```
# petit exemple de script Python
PI = 3.14
nombre = 1          # la variable nombre est initialisée à 1
```

Une expression est une portion de code que Python peut évaluer, composée d'identificateurs, de littéraux et d'opérateurs : `3 * nombre + 5`, `(math.sqrt(3) – 1 ) / 2`, etc.

2.2. Types de données élémentaires

Les types de données prédéfinis qui nous seront utiles sont les types `bool` (booléen), `int` (entier), `float` (flottant) et `str` (chaîne de caractères).

Les expressions de type `bool` peuvent prendre les valeurs `True` ou `False`, et être combinés à l'aide des opérateurs `and`, `or` et `not`. On peut notamment combiner des expressions booléennes utilisant les opérateurs de comparaison : `==, !=, <, >, <=, >= :`

```
monBooleen = (( a > 8 ) and ( a <= 20 )) or (b != 15)
```

Les valeurs des expressions de type `int` ne sont limitées que par la taille mémoire, et celles des expressions de type `float` ont une précision finie. Les littéraux de type `float` sont notés avec un point décimal ou en notation exponentielle : `3.14`, `.09` ou `3.5e8` par exemple.

Les principales opérations définies sur les expressions numériques sont les suivantes :

```
print(17 + 4)       # 21
print(21 – 6.2)     # 14.8
print(3.3 * 2)      # 6.6
print(5 / 2)        # 2.5
print(5 // 2)       # 2 (division entière)
print(5 % 2)        # 1 (modulo)
print(5 ** 2)       # 25 (exponentiation)
print(abs(3-8))     # 5 (valeur absolue)
```

L'import du module `math` donne accès à l'ensemble des fonctions mathématiques usuelles :

```
import math

print(math.cos(math.pi/3))            # 0.5000000000000001 (et oui…)
print(math.factorial(6))              # 720
print(math.log(32,2))                 # 5.0
```

Les littéraux de type `str` sont délimités par des « ' » ou des « " » :

```
chaine1 = "Bonjour"
```

```
chaine2 = ‘coucou les "amis"’
chaine3 = "aujourd’hui"
```

Les chaînes peuvent être manipulées à l’aide des opérations suivantes :

```
chaine1 = ‘bonjour’
print(len(chaine1))             # 7 (len = longueur)

chaine2 = chaine1 + ‘ monde’    # + = concaténation de chaînes
print(chaine2)                  # bonjour monde

chaine2 = chaine1.upper()       # passage en majuscules
print(chaine2)                  # BONJOUR

chaine2 = chaine2.lower()       # passage en minuscules
print(chaine2)                  # bonjour

print(chaine2[0])               # b
print(chaine2[3])               # j
print(chaine2[1:4])             # onj    (de l’indice 1 à l’indice 4
                                #        non compris)
print(chaine2[:3])              # bo     (du début à l’indice
                                #        3 non compris)
print(chaine2[4:])              # our    (de l’indice 4 à la fin)
```

2.3. Affectation et opérations d’entrée-sortie

L’affectation se note à l’aide du symbole ‘=’, qu’il ne faudra pas confondre avec l’opérateur d’égalité, noté ‘==’ :

```
nombre1 = 2 * 18 - 5            # nombre1 ← 31
print(nombre1 == 30)            # False
```

Il est possible de modifier (on dit également « transtyper »), le type d’une expression de la façon suivante :

```
nombre1 = int(2*PI)             # nombre1 ← 6
flottant1 = float(17*2)         # flottant1 ← 34.0
```

Pour saisir une valeur au clavier, on utilise l’instruction `input`, éventuellement complétée par un message à afficher. Il s’agit d’une lecture *en mode texte*, c’est-à-dire que la valeur retournée par cette instruction est de type str ; il est donc la plupart du temps nécessaire de transtyper cette valeur :

```
nbTours = int (input(‘nombre de tours ? ’))   # saisie d’un entier
msg = input(‘message ? ’)                     # saisie d’une chaîne
```

L’instruction `print` (que nous avons déjà plusieurs fois utilisée) permet d’afficher une séquence d’expressions (séparées par des virgules). À l’affichage, un espace est automatiquement inséré entre les différentes valeurs de la liste d’expressions :

```
i = 3
print(‘resultat :’,3*i)      # resultat : 9
```

Par défaut, l’instruction `print` rajoute un saut de ligne après affichage, à moins d’utiliser l’option `end` de la façon suivante :

```
i = 5
print(i,end="")
print(i+1)
                        # affiche 56 avant de changer de ligne
```

Les littéraux de type `str` peuvent contenir des *caractères spéciaux*, préfixés par un ‘\’ (antislash), dont la signification est la suivante :

caractère	signification
\’	apostrophe

\"	guillemet
\n	saut de ligne
\a	bip (sonnerie)
\t	tabulation horizontale

2.4. Structures de contrôle

Rappelons que les instructions ayant la même indentation (même « décalage » par rapport au début de la ligne) appartiennent à un même bloc. Les décalages s'effectuent généralement de 4 en 4, et sont automatiquement gérés dans l'éditeur IDLE.

2.4.1. Alternative simple

La structure `if-else` est l'équivalent Python de notre « `si ... alors ... sinon ... fin_si` » :

```
if a == b:
     print('egalite pour',a,'et',b)
     print('-----------------------')

nombre = 8                    # fin du if…, pas de partie « else »
if nombre > 5:
     print('grand')
else:
     print('petit')

a = 3                         # fin du else…
```

2.4.2. Structure à choix multiple

La structure `if-elif-…-else` est l'équivalent Python de notre « `selon que ... fin_selon` » :

```
note = float(input('Note du baccalaureat : '))
if note >= 16:
     print('mention TB')
elif note >= 14:
     print('mention B')
elif note >= 12:
     print('mention AB')
elif note>= 10:
     print('mention Passable')
else:
     print('collé...')
```

2.4.3. Boucle while

La boucle `while` est l'équivalent Python de notre « `tant que faire ... fin_tantque` » :

```
n = int (input('Donnez un entier compris entre 1 et 10 : ') )
while (n<1) or (n>10):
     print('J\'ai dit compris entre 1 et 10 !')
     n = int (input('Allez, recommencez : ') )
print('Merci !...')
```

Notons que Python ne propose pas d'équivalent de la boucle « `répéter … jusqu'à` »…

2.4.4. Boucle for

La boucle `for` est l'équivalent Python de notre « `pour .. de .. à .. faire .. fin_pour` ». L'expression `range(i)` retourne la liste des entiers allant de `0` à `i` non compris. L'expression `range(i,j)` retourne la liste des entiers allant de `i` à `j` non compris. L'expression `range(i,j,k)` retourne la liste des entiers allant de `i` à `j` non compris *par pas de* `k`.

```
for i in range(7):
    print(2*i,end=' ')      # affiche : 0 1 2 3 4 5 6
for i in range(1,6):
    print(2*i,end=' ')      # affiche : 2 4 6 8 10
for i in range(1,13,3)
    print(i,end=' ')        # affiche : 1 4 7 10
```

2.5. Quelques exemples de scripts Python

Ce premier script calcule le produit de deux nombres en utilisant l'algorithme dit de la « multiplication russe » :

```
# multiplication russe de a par b
# initialisations
a = int(input("a = "))
b = int(input ("b = "))
c = 0;
# boucle de calcul
while (a != 0):
    if (a % 2 == 1):
        c = c + b
    a = a // 2
    b = b * 2
# affichage du résultat
print ("Résultat :", c)
```

Ce deuxième script permet de déterminer si un entier n est ou non premier :

```
# ce script détermine si l'entier N est premier ou non
import math
# lecture de N
N = int(input("N ? "))
# initialisations
racineDeN = int(math.sqrt(N))
diviseur = 2
# tant qu'on n'a pas trouvé de divisieur, on avance...
while ((N % diviseur != 0) and (diviseur <= racineDeN)):
    diviseur = diviseur + 1
# si diviseur est allé au delà de racineDeN, N est premier
if (diviseur > racineDeN):
    print ("Le nombre", N, "est premier.")
else:
    print ("Le nombre", N, "est divisible par", diviseur)
```

Remarque : on pourrait naturellement traiter à part le cas des nombres pairs et ne tester ensuite que les diviseurs 3, 5, 7, 9, 11, etc.

2.6. Traduction d'algorithmes en Python – Tableau de synthèse

Langage algorithmique	Script Python
Algorithme XXX début … fin	*rien de particulier (à notre niveau), le script commence à la première instruction…*
`# ceci est un commentaire`	`# ceci est un commentaire`
`var a, b : entiers`	*pas de déclaration de variables*
`Entrer ( a )`	`a = int(input("valeur de a ? "))`
`Afficher ("résultat : ", res)`	`print ("résultat : ", res)`

a ← a + 2	a = a + 2
quotient division entière	//
reste division entière	%
opérateurs de comparaison : <, >, ≤, ≥, =, ≠	<, >, <=, >=, ==, !=
opérateurs logiques : et, ou, non	and, or, not

si (a = b)
    c = 2 * c
fin_si
	if (a == b):
	  c = 2 * c

si (a = b)
    c = 2 * c
sinon
    a = a + 1
    b = b div 2
fin_si
	if (a == b):
	  c = 2 * c
	else :
	  a = a + 1
	  b = b / 2

tantque (a ≠ 0) Afficher (2 * a) a ← a - 1 fin_tantque	while (a != 0): print 2*a a = a - 1

répéter
    a = 2 * a
    b = b - 1
jusqu'à (b ≤ 0)
	pas de boucle répéter en python, on transforme l'algorithme en utilisant la boucle while :
	a = 2 * a
	b = b - 1
	while (b > 0):
	  a = 2 * a
	  b = b - 1

pour i de 1 à n faire
    a = 3 * i
    Afficher (a)
fin_pour
	for i in range(1,n+1):
	  a = 3 * i
	  print a
	range(a,b) permet de parcourir les valeurs a, a+1, …, b-1 (b non compris)

pour i de 3 à 20 par pas de 2 faire
    a = 3 * i
    Afficher (a)
fin_pour
	for i in range(3,21,2):
	  a = 3 * i
	  print a

Manipulation de chaînes de caractères	- ch[i] : i-ième caractère de ch (les indices démarrent à 0) - len(ch) : nombre de caractères de ch - ch1 + ch2 : concaténation - ch[i:j] : la sous-chaîne de ch allant du i-ième au (j-1)-ième caractère - ch[i:] : la fin de ch, à partir du i-ième caractère - ch[:i] : le début de ch, jusqu'au (i-1)-ième caractère

2.7. Dessiner en Python

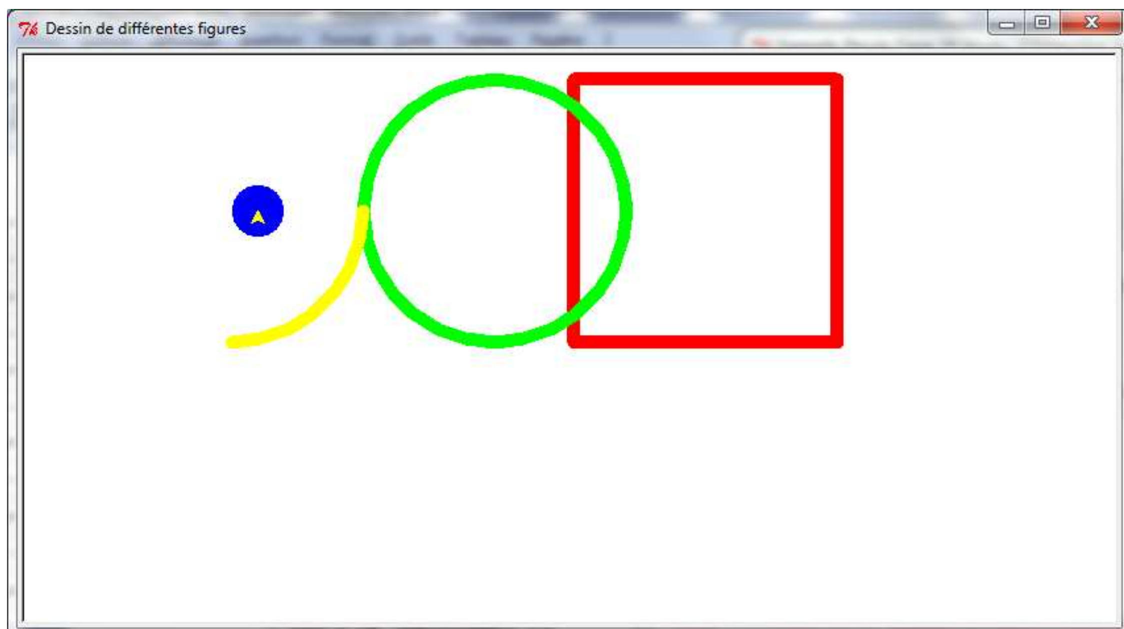
Le module `turtle` permet de dessiner dans une fenêtre graphique à l'aide d'un ensemble de primitives dédiées.

L'exemple suivant, aisément compréhensible, permet de dessiner différentes figures :

```
# script exemple turtle : dessin de différentes figures
import turtle

    # côté du carré
monCote = 200

    # reinitialise la fenêtre
turtle.reset()

    # titre de la fenêtre
turtle.title("Dessin de différentes figures")

    # paramètres de dessin
turtle.color('red')
```

```
turtle.width(10)
turtle.speed(5)

    # dessin d'un carré
turtle.pendown()
for i in range(4):
    turtle.forward(monCote)
    turtle.left(90)

    # partons ailleurs dessiner un cercle vert
turtle.penup()
turtle.goto(-60,0)
turtle.pendown()
turtle.color('green')
turtle.circle(100)

    # puis un arc de cercle jaune
turtle.penup()
turtle.goto(-260,0)
turtle.pendown()
turtle.color('yellow')
turtle.circle(100,90)

    # et enfin un gros point bleu
turtle.penup()
turtle.goto(-240,100)
turtle.pendown()
turtle.dot(40,'blue')
```

Le résultat obtenu est le suivant :



Les principales primitives graphiques qui nous seront utiles sont les suivantes :

PYTHON – Utilisation du module turtle	
import turtle	permet d'importer les fonctionnalités du module turtle
turtle.title(<chaîne>)	donne un titre à la fenêtre turtle (par défaut : Turtle Graphics)
turtle.reset()	efface l'écran, recentre la tortue en (0,0)
turtle.color(<chaîne>)	détermine la couleur du tracé (noir par défaut).

	Couleurs prédéfinies : ‘red’, ‘blue’, ‘green’, etc.
turtle.width(<épaisseur>)	détermine l’épaisseur du tracé
turtle.speed(<vitesse>)	détermine la vitesse du tracé (valeur entière)
turtle.forward(<distance>)	avance d’une distance donnée
turtle.backward(<distance>)	recule d’une distance donnée
turtle.left(<angle>)	tourne à gauche d’un angle donné (exprimé en degrés)
turtle.right(<angle>)	tourne à droite d’un angle donné (exprimé en degrés)
turtle.circle(<rayon>{,<angle>})	dessine un cercle de rayon donné, ou un arc de cercle de rayon et angle donnés
turtle.dot(<diamètre>,<couleur>)	dessine un point (cercle plein) de diamètre et couleur donnés
turtle.penup()	relève le crayon (pour pouvoir avancer sans dessiner)
turtle.pendown()	abaisse le crayon (pour pouvoir se déplacer en dessinant)
turtle.goto(x,y)	se déplace jusqu’au point de coordonnées (x,y)
turtle.xcor(), turtle.ycor()	retourne la coordonnée courante (abscisse ou ordonnée) de la tortue
turtle.write(<chaîne>)	écrit la chaîne de caractères

Remarque. Il peut arriver que la fenêtre graphique, une fois le programme terminé, ne réponde pas… Pour remédier à ce problème, il suffit que le programme idle.pyw soit lancé avec l’option -n. Une façon simple de réaliser cela est d’effectuer une copie idle2.bat du fichier idle.bat (situé dans le répertoire Lib\idlelib de l’installation Python), et de le modifier ainsi :

Fichier idle.bat (à modifier) :

```
@echo off
rem Start IDLE using the appropriate Python interpreter
set CURRDIR=%~dp0
start "%CURRDIR%..\..\pythonw.exe" "%CURRDIR%idle.pyw" %1 %2 %3 %4 %5 %6 %7 %8 %9
```

Fichier idle2.bat (modifié) :

```
@echo off
rem Start IDLE using the appropriate Python interpreter
set CURRDIR=%~dp0
start "%CURRDIR%..\..\pythonw.exe" "%CURRDIR%idle.pyw" -n %1 %2 %3 %4 %5 %6 %7 %8 %9
```

Il suffit alors de lancer idle2.bat au lieu de idle.bat…

Chapitre 3. Pour aller (un petit peu) plus loin…

Nous mentionnons dans cette partie quels éléments complémentaires, qui ne rentrent pas dans le cadre du programme de la classe de seconde (sans toutefois aborder des questions de programmation objet, qui se situent hors du cadre de cet enseignement).

3.1. Nombres complexes

Python offre le type numérique `complex` (en notation cartésienne avec deux flottants, la partie imaginaire étant suffixée par `j` : `3.2 + 5j` par exemple), qui s'utilise ainsi :

```
print(6j)                    # 6j
print(3.4+1.2j)              # (3.4+1.2j)
print((3.4+1.2j).real)       # 3.4
print((3.4+1.2j).imag)       # 1.2
print(abs(3+9j))             # 9.486832980505138 (module)
```

Le module `cmath` donne accès à d'autres primitives (pour plus de détails, voir http://docs.python.org/py3k/library/cmath.html#module-cmath).

3.2. Listes

Une liste est une collection ordonnée d'éléments, éventuellement de nature distincte :

```
jours = ['lun', 'mar', 'mer', 'jeu', 'ven', 'sam', 'dim']
pairs = [0, 2, 4, 6, 8]
reponses = ['o', 'O', 'n', 'N']
listeBizarre = [jours, 2, 'hello', 54.8, 2+7j]
```

Les éléments sont repérés par leur numéro d'ordre au sein de la liste (ces numéros démarrent à 0) :

```
liste = [ 2, 3, 4 ]
print(liste)                 # [2, 3, 4]
print(liste[1])              # 3
liste[1] = 28
print(liste)                 # [2, 28, 3]
```

D'autres possibilités de manipulation des listes :

```
liste = list(range(4))       # convertit en liste
print(liste)                 # [0, 1, 2, 3]

print(1 in liste)            # True
print(5 in liste)            # False

liste = [6, 8, 1, 4]
liste.sort()                 # tri de la liste
print(liste)                 # [1, 4, 6, 8]

liste.append(14)
print(liste)                 # [1, 4, 6, 8, 14]

liste.reverse()
print(liste)                 # [14, 8, 6, 4, 1]

liste.remove(8)
print(liste)                 # [14, 6, 4, 1]

i = liste.pop()              # récupère et supprime le dernier élément
print(i)                     # 1
print(liste)                 # [14, 6, 4]

liste.extend([6,5])
```

```
print(liste)                # [14, 6, 4, 6, 5]
print(liste.count(6))       # 2
print(liste[1:3])           # [6, 4]      position 1 à 3 non compris
liste[0:2] = [5,4,3]        # remplace une portion de liste
print(liste)                # [5, 4, 3, 4, 6, 5]
liste[4:] = []
print(liste)                # [5, 4, 3, 4]
liste[:2] = [1,1,1,1]
print(liste)                # [1, 1, 1, 1, 3, 4]
```

3.3. Fonctions

Les fonctions permettent de décomposer une tâche en tâches « plus simples » et souvent d'éviter des répétitions de portions de code. Elle permettent par ailleurs la *réutilisation* de code (en recopiant la fonction d'un script à un autre ou, plus efficacement, en utilisant le mécanisme d'*import*).

Le format général de définition d'une fonction est le suivant :

```
def nomFonction(paramètres):
    """Documentation de la fonction."""
    <bloc_instructions>
```

La partie « documentation », bien que facultative, est naturellement fortement conseillée (pensons aux réutilisations ultérieures).

Voici un exemple de fonction simple :

```
def maximum(a,b) :
    """détermine le maximum des valeurs a et b """
    if a>b :
        return a
    else :
        return b
```

et quelques exemples d'utilisation (appel) :

```
print(maximum(6,21))                # 21
print(maximum(16,2))                # 16
print(maximum('bonjour','salut'))   # 'salut'
print(maximum([6,1],[4,11,3]))      # [6,1]
```

On voit donc que, du fait de l'utilisation de l'opérateur '<' dans la fonction `maximum`, celle-ci est utilisable sur tous les types de données *ordonnables* (entiers, flottants, chaînes, listes, mais pas nombre complexe !). Dans le cas des listes, il s'agit de l'ordre lexicographique : comparaison des deux premiers éléments puis, en cas d'égalité, des deux suivants, etc.

Le passage de paramètres s'effectue *par affectation*. Ainsi, lors de l'appel

```
m = maximum(x,18)
```

les deux affectations « `a=x` » et « `b=18` » sont réalisées avant que le corps de la fonction `maximum` s'exécute.

On peut définir des fonctions n'utilisant pas l'instruction `return`. Elles correspondent à des actions sans paramètre résultat :

```
def afficheMultiple(a,n) :
    """affiche les n premiers multiples non nuls de a"""
    for i in range(1,n+1) :
    print(i*a,end=" ")
```

On aura ainsi :

```
afficheMultiple(6,5)        # 6 12 18 24 30
afficheMultiple(1,4)        # 1 2 3 4
```

On peut également définir des fonctions ayant plusieurs résultats, en utilisant des `return` *multiples* :

```
def divisionEuclidienne(a,b) :
     """calcule le reste et le quotient de la division de a par b"""
     return a//b, a%b
```

qui s'utilise ainsi :

```
q,r = divisionEuclidienne(17,4)  # q reçoit le quotient, r le reste
print(q)                         # 4
print(r)                         # 1
```

Les paramètres d'une fonction sont des paramètres d'*entrée*, non modifiables par la fonction. Si une fonction a besoin de modifier des paramètres (paramètres d'*entrée-sortie*), il faut que ceux-ci soient à la fois paramètres d'entrée et paramètres de sortie (renvoyés par `return`), et d'appeler cette fonction sous une forme adéquate.

Ainsi, l'extrait suivant :

```
def rallonge(liste) :
     """rajoute un entier lu en fin de liste"""
     n = int(input('entier ? '))
     return liste+[n]

maListe=[1, 6]
maListe=rallonge(maListe)
print(maListe)     # [1, 6, 9] (si l'entier 9 a été entré au clavier)
```

permet de rallonger la liste `maListe`… Notons qu'une instruction telle que

```
print(rallonge(maListe))
```

afficherait la liste `[1, 6, 9]` (en supposons que nous entrons à nouveau l'entier `9` au clavier), mais ne modifierait pas la liste `maListe` dont le contenu serait toujours `[1, 6]`.

3.4. Visibilité des variables

Une variable définie (c'est-à-dire affectée) dans une fonction n'est visible qu'à l'intérieur de cette fonction, c'est une variable *locale*.

Une variable définie dans un script à l'extérieur de toute fonction est une variable *globale*. Elle est visible (c'est-à-dire que sa valeur est utilisable) partout, y compris dans les fonctions définies dans le script.

Par contre, une fonction ne peut pas modifier la valeur d'une variable globale… En effet, une instruction modifiant la valeur d'une telle variable dans une fonction fait que cette variable est alors considérée comme locale, et donc distincte de la variable globale portant le même nom…