Python Turtle

Release 2020

Apr 17, 2020

Contents

Dans ce cours tu apprends à programmer en Python en utilisant une tortue pour faire des dessins. Mais le message principale c'est:

- Programme toi-même (évite le copier-coller)
- Modifie, essaye, invente

CHAPTER 1

Introduction

Dans ce tutoriel, tu vas apprendre à programmer dans un langage qui s'appelle **Python**. Tu vas programmer les déplacements d'une tortue. Voici à quoi ça va ressembler.

`epfl1.py`

Cette tortue laisse une trace qui te permet de faire des dessins. Mais tout d'abord, tu dois télécharger un outil pour programmer.

1.1 Installer un éditeur

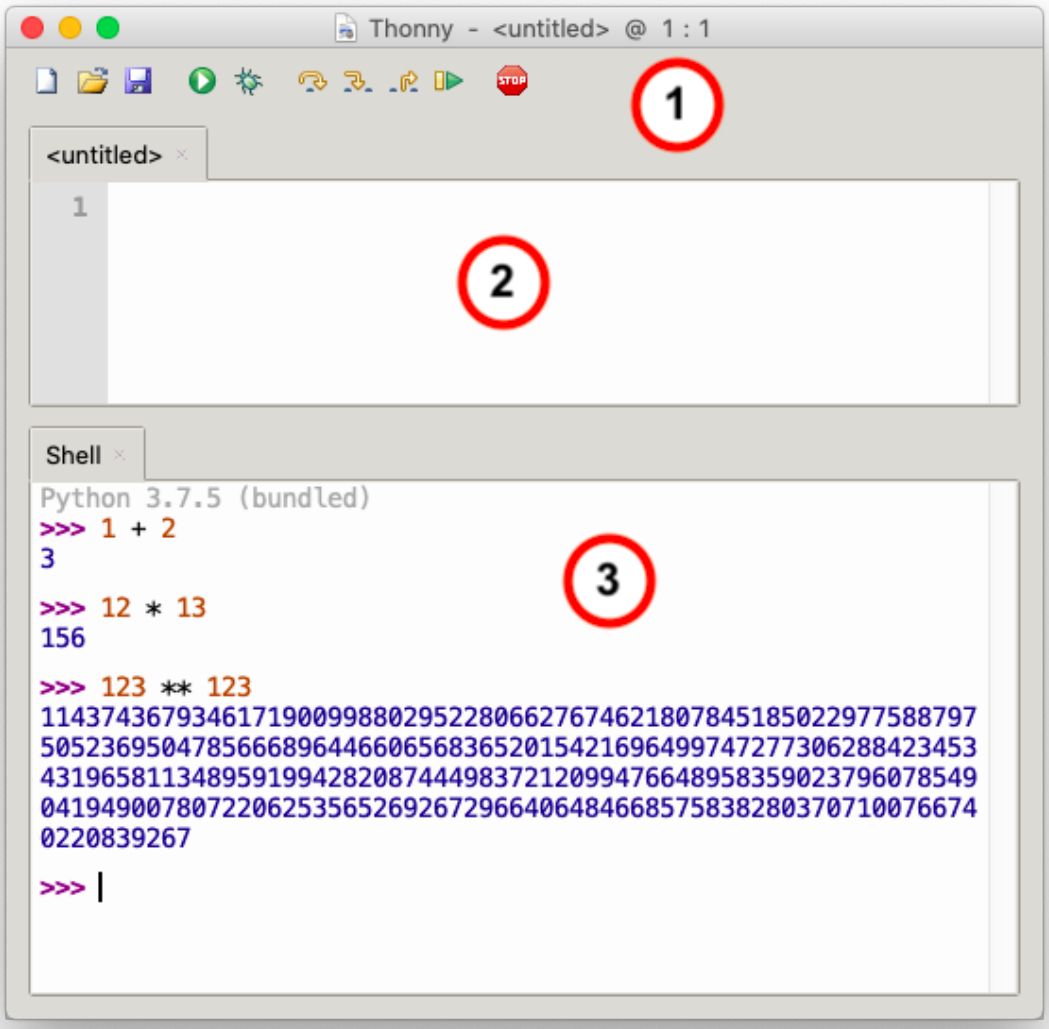
Va sur le site https://thonny.org et télécharge l'application **Thonny**.

C'est un éditeur de programme qui te permet:

- d'**écrire** un programme
- d'**exécuter** ce programme
- d'**afficher** le résultat

Dans une fenêtre de Thonny tu as trois régions:

1. les **boutons** pour *Créer, Ouvrir, Sauvegarder, Exécuter* un programme
2. la partie **éditeur** pour écrire un programme entier
3. la **console** pour executer des commandes courtes (Shell)



1.2 Les premiers pas

Dans la console (ou Shell), tu peux directement entrer des expressions courtes que Python va évaluer. Après les 3 chevrons (>>>) tu peux écrire cette addition:

```
>>> 1 + 2
3
```

Tu peux aussi essayer cette muliplication:

```
>>> 12 * 13
156
```

Python n'est pas limité dans le nombre de chiffres qu'un calcul peut produire. Voici une puissance (**) de deux nombres qui donne un résultat qui s'étale sur 5 lignes:

```
>>> 123 ** 123
1143743679346171900998802952280662767462180784518502297758879
7
5052369504785666896446606568365201542169649974727730628842345
3
4319658113489591994282087444983721209947664895835902379607854
9
0419490078072206253565269267296640648466857583828037071007667
4
0220839267
```

1.3 Dessiner avec une tortue

Dans cet exercice, on va s'entrainer en n'utilisant que la console, donc la partie basse de la fenêtre. Cela te permet de voir directement l'effet d'une commande.

Par la suite, nous allons utiliser le module `turtle`. Ce module te permet de déplacer une tortue sur l'écran, à l'aide des commandes que tu programmes.

Pour pouvoir utiliser ce module, tu dois l'importer au début du programme:

```
>>> from turtle import *
```

Ensuite tu peux donner un ordre à ta tortue:

```
>>> forward(200)
```

Cette commande fait avancer la tortue de 200 pixels. Une commande pour contrôler la tortue a la forme suivante:

- une commande (forward, backward, left, right, etc.)
- des parenthèses `( )`
- un argument numériques (distance, angle)

Par exemple, pour faire reculer la tortue de 200 pixels, écris ceci:

```
>>> backward(200)
```

Pour faire tourner la tortue de 90 degrés vers la gauche:

```
>>> left(90)
```

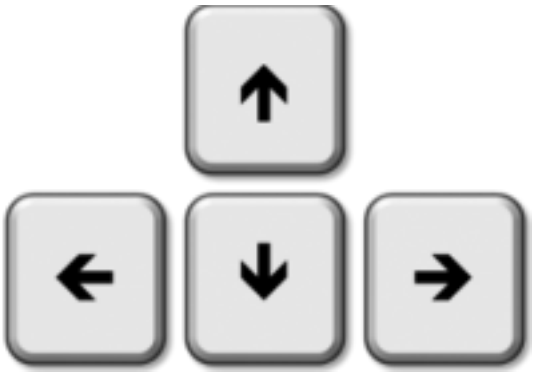
Pour faire tourner la tortue de 45 degrés vers la droite:

```
>>> right(45)
```

La console est un outil très pratique. A n'importe quel moment tu peux facilement tester des commandes.

1.4 Remonter dans l'historique

Toutes les commandes que tu as entrées dans la console restent en mémoire. Tu peux facilement les retrouver, les modifier et les réutiliser.



Il suffit d'utiliser les flèches **haut** et **bas** pour te balader dans l'historique de tes commandes.

Essaye, c'est très pratique.

1.5 Ecrire un programme

Toutes ces commandes que tu peux écrire directement dans la console, tu peux aussi les mettre dans un **programme** (qu'on appelle **script**).

Un programme n'est rien d'autre qu'une *liste de commandes qui disent à la tortue comment bouger*. Une fois le programme terminé, tu peux l'exécuter à l'aide du bouton vert **Exécuter**. À ce moment, Thonny te demande de donner un nom à ton programme.

Le programme suivant fait dessiner un triangle à la tortue. Essaye de le programmer!

```
from turtle import *

forward(200)
left(120)
forward(200)
```

(continues on next page)

(continued from previous page)

```
left(120)
forward(200)
left(120)

done()
```

triangle.py

L'instruction `done()` doit toujours être la dernière instruction d'un programme. Elle est nécessaire pour garder le fenêtre ouverte jusqu'à ce que tu cliques sur le bouton de fermeture de fenêtre.

1.6 Dessiner une maison

En utilisant des angles de 45 et 90 degrés, tu peux dessiner une maison.

```
from turtle import *

forward(141)
left(90)
forward(100)
left(45)
forward(100)
left(90)
forward(100)
left(45)
forward(100)

done()
```

house.py

1.7 Ajouter une porte

Tu peux ajouter une porte en dessinant encore un rectangle.

```
from turtle import *

forward(141)
left(90)
forward(100)
left(45)
forward(100)
left(90)
forward(100)
left(45)
forward(100)

# draw a door
left(90)
```

(continues on next page)

(continued from previous page)

```
forward(50)
left(90)
forward(50)
right(90)
forward(30)
right(90)
forward(50)
left(90)
forward(200)

done()
```

house2.py

1.8 Erreurs fréquentes

Si tu fais une erreur avec le nom du module, par exemple *turtel* au lieu de *turtle*, tu obtiens une erreur de type `ModuleNotFoundError`:

```
>>> from turtel import *
ModuleNotFoundError: No module named 'turtel'
```

Si tu oublies de donner un argument à une fonction, par exemple si tu oublies de mettre und distance pour la fonction *forward()*, tu obtiens une erreur de type `TypeError`:

```
>>> forward()
TypeError: forward() missing 1 required positional argument: 'distance'
```

Si tu fais une erreur dans le nom d'une fonction, par exemple *foreward* au lieu de *forward*, tu obtiens une erreur de type `AttributeError`:

```
>>> foreward(100)
AttributeError: module 'turtle' has no attribute 'foreward'
```

Conseil

- Lis la dernière ligne du message d'erreur
- Essaie de trouver l'erreur dans ton code
- Corrige l'erreur et relance ton programme

1.9 Raccourcir les commandes

Si tu veux écrire tes commandes encore plus courtes, tu peux utiliser les raccourcis à deux lettres:

- `fd` (forward)
- `bk` (back)
- `lt` (left)
- `rt` (right)

Le programme pour dessiner une maison se réduit alors à:

```
fd(141)
lt(90)
fd(100)
lt(45)
fd(100)
lt(90)
fd(100)
lt(45)
fd(100)
```

CHAPTER 2

Dessiner

Tu as vu les 4 commandes de base pour déplacer la tortue: *forward, backward, left* et *right*. Mais tu peux contrôler d'autres aspects du dessin:

- épaisseur du trait
- couleur du trait
- couleur de remplissage

En plus tu peux aussi:

- monter et descendre le stylo
- dessiner des cercles
- ajouter du texte

2.1 Monter le stylo

La tortue peut monter et descendre son stylo. Ceci lui permet de dessiner des lignes séparées:

```
from turtle import *

forward(100)
up()
forward(50)
down()
forward(100)

done()
```

draw1.py

2.2 Téléporter la tortue

La tortue peut aller directement (en ligne droite) à n'importer quel position indiquée par des coordonnés (x, y). La commande `goto(0, 20)` va téléporter la tortue à la position (x=0, y=20).

```
from turtle import *

forward(200)

up()
goto(0, 20)
down()
forward(200)

up()
goto(0, 40)
down()
forward(200)

done()
```

draw2.py

2.3 L'épaisseur de ligne

Tu peux modifier l'épaisseur du stylo avec la commande `width(2)`.

```
from turtle import *

forward(200)

up()
goto(0, 20)
down()
width(2)
forward(200)

up()
goto(0, 40)
down()
width(5)
forward(200)

done()
```

draw3.py

2.4 La couleur de ligne

Tu peux modifier la couleur du stylo, par exemple en blue, avec la commande `pencolor('blue')`.

Voici les couleurs que tu peux choisir:

yellow, gold, orange, red, maroon, violet, pink, magenta, purple, navy, blue, sky blue, cyan, turquoise, lightgreen, green, darkgreen, chocolate, brown, black, gray, white

Tu en trouve une centaine de couleurs sur ce site: http://cng.seas.rochester.edu/CNG/docs/x11color.html

Fais attention de mettre le nom de la couleur entre apostrophes. Par exemple:

- `'Pink'`
- `'HotPink'`
- `'DeepPink'`
- `'Fuchsia'`

```
from turtle import *

width(8)
forward(200)
left(90)

pencolor('pink')
forward(100)
left(90)

pencolor('fuchsia')
forward(300)

done()
```

`draw4.py`

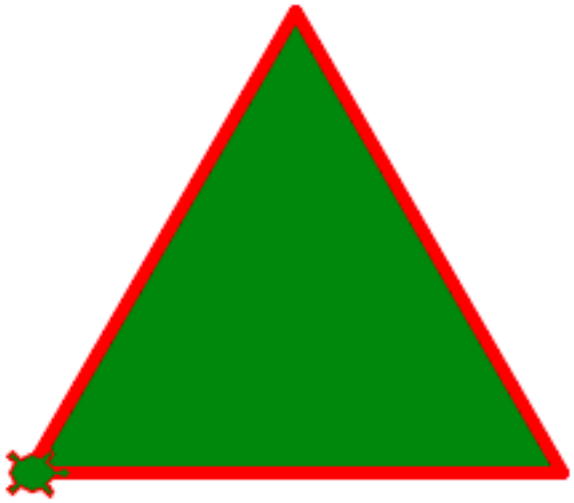
2.5 Remplir une forme

Il est possible de remplir une forme d'une couleur.

Avec la commande `fillcolor('green')` tu peux choisir une couleur de remplissage.

Ensuite il faut entourer la forme à remplir, par exemple le triangle, avec les deux commandes:

```
begin_fill()
end_fill()
```



```
from turtle import *

width(5)
fillcolor('green')

begin_fill()
forward(200)
left(120)
forward(200)
left(120)
forward(200)
left(120)
end_fill()

done()
```

draw5.py

2.6 Ajouter un texte

Il est possible d'écrire un texte à la position de la tortue avec la commande `write()`. Cette commande peut avoir plusieurs arguments. Nous allons en voir deux: le premier est le texte que tu veux écrire (à mettre entre apostrophe), le deuxième définit la police du texte, ainsi que la taille de la police:

```
write('texte', font=('police', taille))
```

La taille de la police par défaut est très petite. Il est préférable que tu l'augmentes.


```
from turtle import *

left(90)
write('default text size')

forward(30)
write('Courier 24', font=('Courier', 24))

forward(30)
write('Arial 36', font=('Arial', 36))

done()
```

draw6.py

2.7 Dessiner un cercle

La fonction `circle(40)` permet de dessiner un cercle avec un rayon de 40 pixels. Dans le programme ci-dessous, la tortue dessine deux cercles.

```
from turtle import *

forward(100)
circle(40)
forward(100)
circle(40)
forward(100)

done()
```

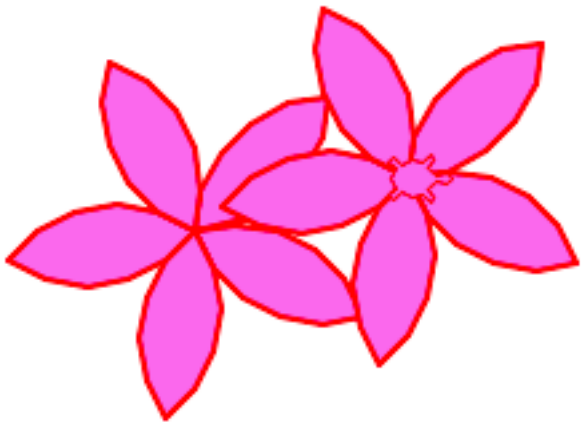
draw7.py

CHAPTER 3

La fonction

Avec les 4 fonctions *forward, backward, left, right* tu peux tout dessiner. Par contre, dès que ton dessin devient un peu plus complexe ton programme

- comporte beaucoup de répétitions,
- est difficile à lire (et comprendre).



La solution est d'utiliser une **fonction** qui permet de

- mettre un bout du code à part
- donner un **nom** à ce code
- **réutiliser** ce code autant de fois que tu veux

Les deux fleurs ci-dessus ont été dessinées à l'aide d'une fonction qu'on a appelé `flower()`.

3.1 Définir une fonction

Tout d'abord nous devons définir la fonction. Une définition de fonction est constituée :

- du **mot-clé** `def` (qui veut dire définition)
- du **nom** de la fonction
- de **parenthèses** `()`
- du signe **deux-points** `:`
- du **corps** de la fonction contenant le code.

Voici, par example, la définition de la fonction **triangle**:

```
def triangle():
    forward(100)
    left(120)
    forward(100)
    left(120)
    forward(100)
    left(120)
    forward(100)
```

Pour indiquer les commandes qui font partie de cette fonction, tu dois les décaler vers la droite. On appelle ça **indenter** le corps de la fonction.

Cette **indentation** est un point important qui distingue Python d'autres langages de programmation. L'indentation te montre visuellement ce qui fait partie du corps de la fonction.

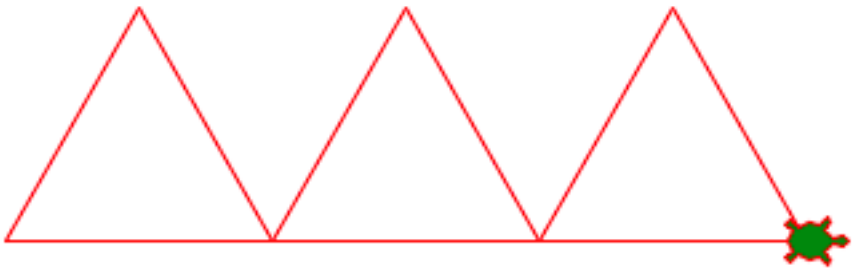
Quand une expression en Python se termine avec le signe **deux-points**, la ligne suivante est automatiquement indenté de 4 espaces.

La définition d'une fonction toute seule ne vas rien dessiner. Tu dois **appeler** la fonction dans ton programme pour exécuter son code. Pour appeler une fonction dans ton programme tu dois écrire:

- le **nom** de la fonction
- des **parenthèses** `()`

Voici la fonction `triangle()` appelée trois fois:

```
triangle()
triangle()
triangle()
```



```
from turtle import *
```

(continues on next page)

(continued from previous page)

```
def triangle():
    forward(100)
    left(120)
    forward(100)
    left(120)
    forward(100)
    left(120)
    forward(100)

triangle()
triangle()
triangle()

done()
```

`func1.py`

Tu dois d'abord définir une fonction avant que tu puisse l'appeler. C'est pour cela qu'on met les définitions de fonction au début du programme. Une fois définie, tu peux appeler une fonction autant de fois que tu veux.

3.2 Appeler une fonction dans une fonction

Une fonction peut appeler une autre fonction. Pour dessiner un carré, nous pouvons d'abord définir une fonction `side` qui dessine juste un côté et tourne de 90 degrées:

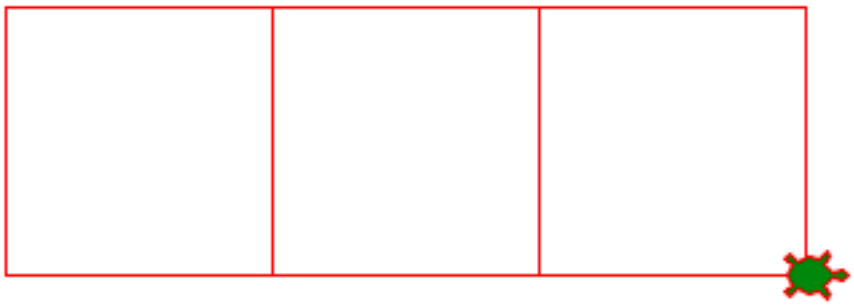
```
def side():
    forward(100)
    left(90)
```

Ensuite la fonction `square()` appelle 4 fois cette fonction:

```
def square():
    side()
    side()
    side()
    side()
    forward(100)
```

Et finalement tu peux appeler 3 fois la fonction `square`:

```
square()
square()
square()
```



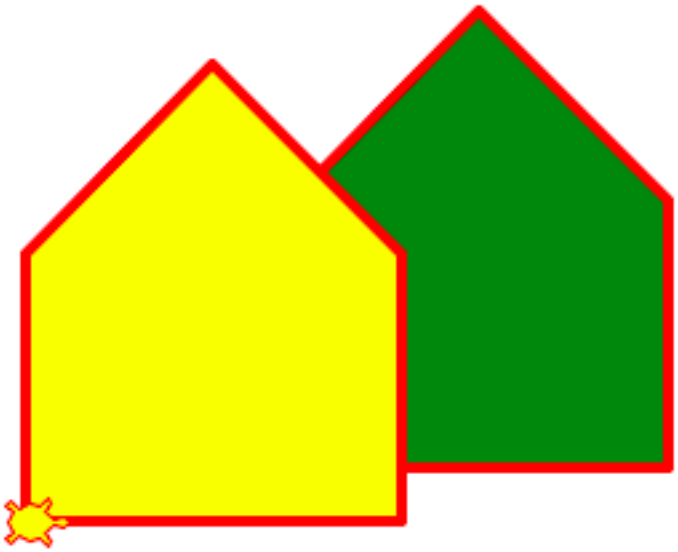
```
from turtle import *

def side():
    forward(100)
    left(90)

def square():
    side()
    side()
    side()
    side()
    forward(100)

square()
square()
square()

done()
```

func2.py

3.3 Dessiner des maisons

Tu peux reprendre le dessin de la maison vu dans l'introduction. Nous allons le définir comme fonction. En plus, nous allons ajouter un remplissage. Ceci va nous permettre de dessiner plusieurs maisons à des positions et dans les couleurs que tu veux.

Voici un exemple:



```
from turtle import *

def house():
    begin_fill()
    forward(141)
    left(90)
    forward(100)
    left(45)
    forward(100)
    left(90)
    forward(100)
    left(45)
    forward(100)
    left(90)
    end_fill()

width(4)
house()
goto(-100, -20)
fillcolor('yellow')
house()

done()
```

house.py

3.4 Dessiner des fleurs

Tu peux utiliser deux arcs de cercle de 90 degrés pour dessiner un pétale:

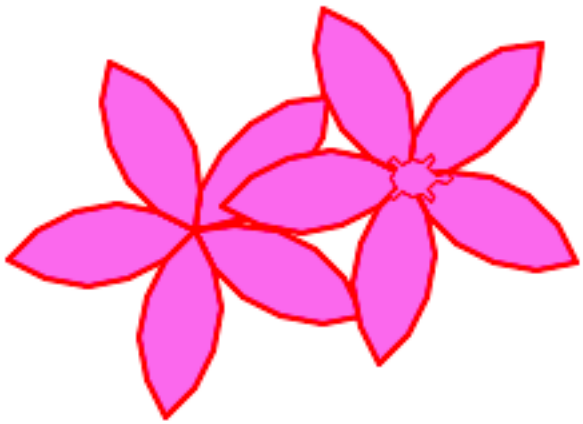
```
def petal():
    begin_fill()
    circle(50, 90)
    left(90)
```

(continues on next page)

(continued from previous page)

```
    circle(50, 90)
    end_fill()
    left(18)
```

Ensuite tu peux combiner 5 pétales pour dessiner une fleur.



```
from turtle import *

def petal():
    begin_fill()
    circle(50, 90)
    left(90)
    circle(50, 90)
    end_fill()
    left(18)

def flower():
    petal()
    petal()
    petal()
    petal()
    petal()

width(2)
fillcolor('violet')
flower()
goto(80, 20)
flower()

done()
```

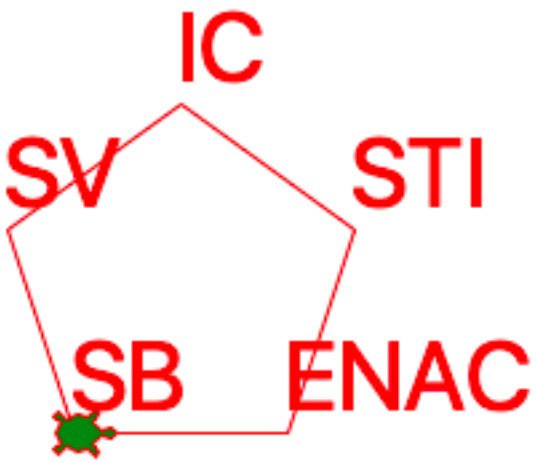
flower.py

CHAPTER 4

La variable

Parfois nous utilisons la même valeur numérique plusieurs fois dans un programme. Il est alors pratique de pouvoir donner un nom à cette valeur. Une **variable** permet d'associer un **nom** à une **valeur**.

L'exemple ci-dessous montre la tortue qui visite les 5 facultés de l'EPFL. Les noms des facultés sont stockés dans une `variable`.



`epfl2.py`

4.1 Dessiner un rectangle

Un rectangle est défini entièrement par deux grandeurs:

- la largeur
- la hauteur

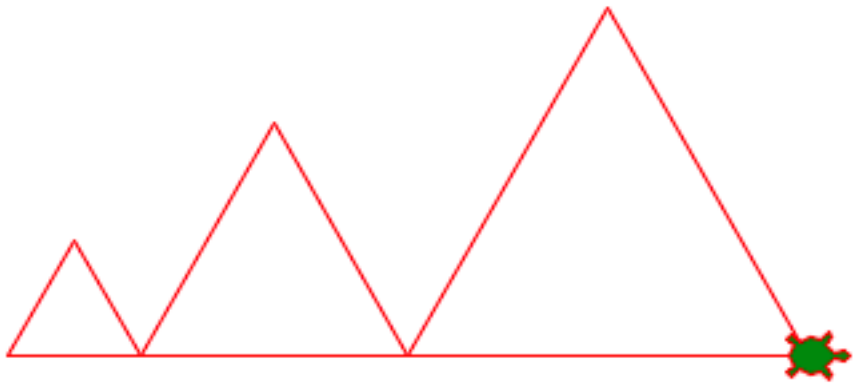
Quand on dessine un rectangle, chaque valeur est utilisée 2 fois. Tu peux définir ces valeurs au début du programme avec des variables. Dans le programme tu peut ensuite utiliser ces variables à la places de valeurs numériques.

```
from turtle import *

a = 200
b = 100

forward(a)
left(90)
forward(b)
left(90)
forward(a)
left(90)
forward(b)
left(90)

done()
```

var1.py

4.2 Changer la valeur d'une variable

À n'importe quel moment dans un programme, tu peux changer la valeur d'une variable. Dans l'exemple qui suit, nous mettons d'abord la valeur de la variable `a` à 50. Ensuite nous la changeons à 100 et finalement à 150. A chaque fois nous dessinons un triangle qui utilise la valeur de cette variable comme longueur.



```
from turtle import *

def triangle():
    forward(a)
    left(120)
    forward(a)
    left(120)
    forward(a)
    left(120)
    forward(a)

a = 50
triangle()
a = 100
triangle()
a = 150
triangle()

done()
```

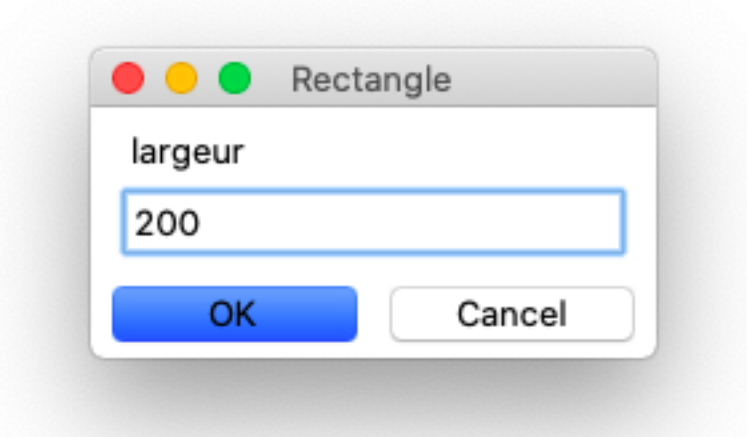
var2.py

4.3 Demander une valeur

Python permet de demander une valeur à l'utilisateur en utilisant une des fonctions prédéfinies:

```
a = numinput('Rectangle', 'largeur')
```

Cette fonction va ouvrir une fenêtre de dialogue avec le titre **Rectangle** et un texte pour le paramètre dont il faut donner une valeur, ici on va afficher le texte **largeur**.



Tu peux entrer un nombre, par exemple 200. Quand tu cliques sur **OK** ce nombre va être mis dans la variable `a` .

Le programme suivant demande également la hauteur du rectangle et met cette valeur dans la variable `b`.

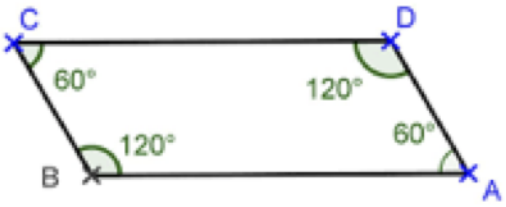
```
from turtle import *

a = numinput('Rectangle', 'largeur')
b = numinput('Rectangle', 'hauteur')

forward(a)
left(90)
forward(b)
left(90)
forward(a)
left(90)
forward(b)
left(90)

done()
```

Une fois les deux valeurs obtenues, la tortue commence à dessiner le rectangle.

`var3.py`

4.4 Dessiner un parallélogramme

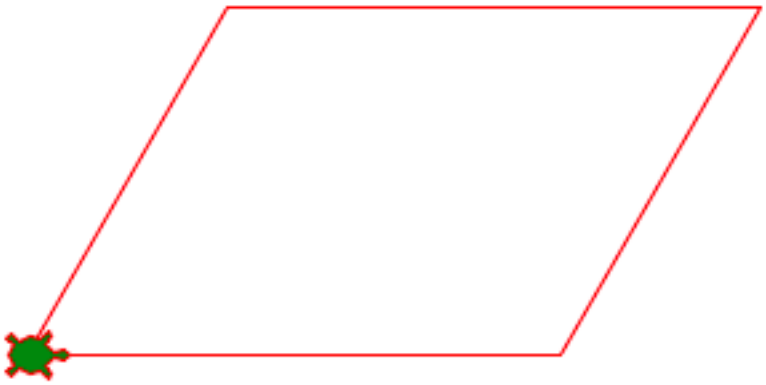
Deux angles voisins d'un parallélogramme se complètent pour donner 180 degrés.



Nous pouvons donc calculer l'angle complémentaire selon l'expression:

```
left(180-angle)
```

Il est facile de changer les angles du parallélogramme quand on utilise une variable. Au lieu de changer 4 valeurs, on modifie une seule valeur au début du programme. Les 4 angles sont calculés par la suite en utilisant cette variable.



```
from turtle import *

a = 200
b = 150
angle = 60

forward(a)
left(angle)
forward(b)
left(180-angle)
forward(a)
left(angle)
forward(b)
left(180-angle)

done()
```

var4.py

CHAPTER 5

La boucle

Dans un programme, il a souvent des bouts de code qui doivent être répétés. On utilise alors une structure qu'on appelle **boucle** pour indiquer au programme de répéter certaines instructions.

Le nouveau logo de l'EPFL utilise des grands carrés rouges. Ci-dessous, la tortue dessine dans une boucle les 4 côtés du carré rouge. Ensuite la tortue répète dans une deuxième boucle ces carrés 5 fois.

`epfl3.py`

5.1 Dessiner un carré

On peut dessiner un carré en répétant 4 fois ces instructions

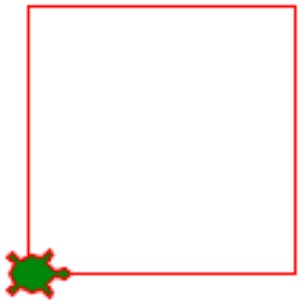
```
from turtle import *

forward(100)
left(90)
forward(100)
left(90)
```

(continues on next page)

(continued from previous page)

```
forward(100)
left(90)
forward(100)
left(90)

done()
```

square1.py



La **boucle** est une structure qui permet de répéter des lignes de code. Pour répéter un bout de code un certain nombre de fois (ici 10 fois par exemple), on écrit:

```
for i in range(10):
    code
    ...
```

Cette structure est appelée la **boucle for**. Elle se compose des éléments suivants:

- le mot-clé `for`
- une variable, souvent appelée `i`
- le mot-clé `in`
- la fonction `range(n)`
- le signe deux-points `:`

En Python, le signe *deux-points* est toujours suivi de quelques lignes code indenté. Normalement les lignes de code qui suivent sont décalées 4 espaces vers la droite.

Le programme du carrée devient beaucoup plus compact si on utilise une boucle. Au lieu de 8 lignes, on n'auras besoin que de 3 lignes.

```
from turtle import *

for i in range(4):
    forward(100)
    left(90)

done()
```

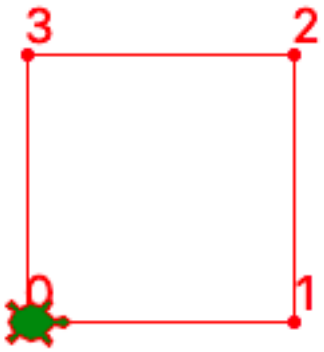
square2.py

5.2 Le compteur de boucle

Le `i` dans l'expression de boucle `for i in range(4)` est bien une variable. En fait c'est une variable qui va prendre successivement les valeurs 0, 1, 2, 3.

En Python, comme dans la programmation en général, on a l'habitude de toujours commencer à compter à 0. C'est la raison pour laquelle on s'arrête à 3, et non pas à 4 dans `range(4)`. Ce sont bien 4 répétitions de la boucle: 0, 1, 2, et 3.

Nous allons utiliser la fonction `write(i)` pour écrire cette valeur du compteur de boucle à chaque sommet du carré.




```
from turtle import *

for i in range(4):
    write(i, font=(None, '18'))
    forward(100)
    left(90)
    dot()

done()
```

`square3.py`

5.3 Une boucle dans une boucle

Tu peux même mettre une boucle dans une boucle. On appelle ça des **boucles imbriquées**.

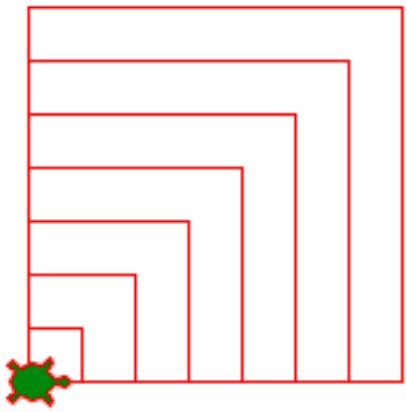
Dans l'exemple suivant, tu as une première boucle qui se répète 7 fois. Mais à l'intérieur il y une deuxième boucle qui dessine un carré. La longueur du carré est donnée par la variable `a`. Cette valeur est initialisée avant la boucle avec:

```
a = 20
```

Après chaque passage de boucle cette valeur est augmentée de 20:

```
a += 20
```

L'opérateur += est un raccourci pour dire `a = a + 20`



```
from turtle import *

a = 20
for i in range(7):
    for j in range(4):
        forward(a)
        left(90)
    a += 20

done()
```

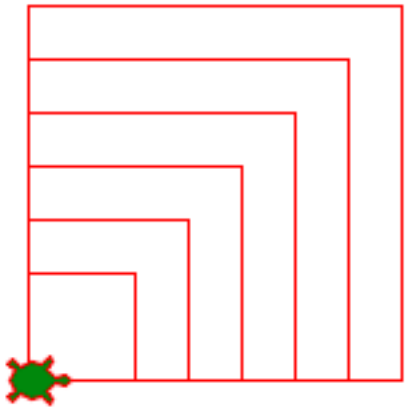
`square4.py`

Une façon plus compacte est d'utiliser un indicateur de plage de la forme suivante:

```
for a in range(40, 160, 20):
```

- la valeur 40 est la valeur initiale (au premier passage: a=40)
- la valeur 160 la valeur finale (mais pas atteint, au dernier passage a<160)
- la valeur 20 est l'incrément (on ajoute 20 à chaque passage)

La variable `a` va donc prendre successivement les valeurs 40, 60, 80, 100, 120, et 140.



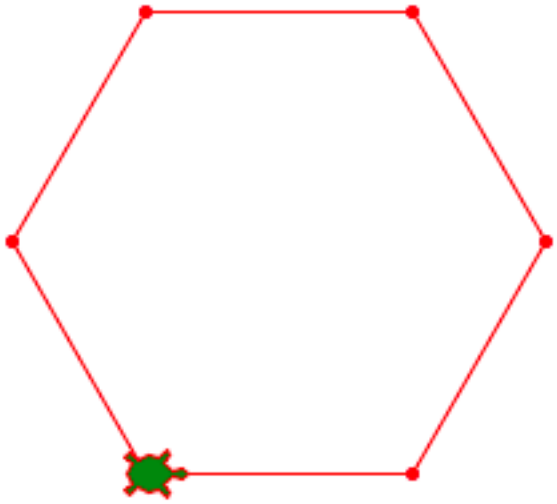
```
from turtle import *

for a in range(40, 160, 20):
    for i in range(4):
        forward(a)
        left(90)

done()
```

square5.py

5.4 Dessiner un polygone

En utilisant une boucle tu peux très facilement programmer ta tortue pour dessiner un polygone. Si le polygone possède `n = 6` sommet, la tortue doit tourner à chaque sommet:

```
left(360/n)
```



```
from turtle import *

n = 6
for i in range(n):
    forward(100)
    dot()
    left(360/n)

done()
```

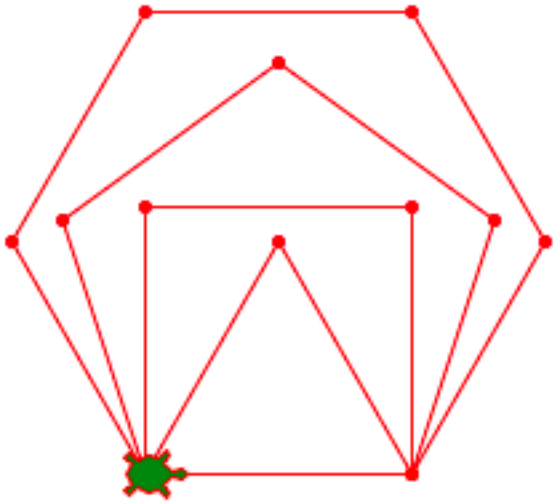
polygon1.py

5.5 Dessiner plusieurs polygones

De nouvea,u tu peux imbriquer deux boucles l'une dans l'autre pour dessiner plusieurs polygones. Cette fois nous utilisons l'expression de boucle suivante:

```
for n in range(3, 7):
```

Le compteur de boucle `n` ne commence cette fois pas à 0 mais à 3. Il va parcourir successivement les valeurs 3, 4, 5, et 6 (il doit rester strictement inférieur à 7). Dans la boucle intérieure la tortue va donc dessiner un triangle, un carré, un pentagone et un hexagone.



```
from turtle import *

for n in range(3, 7):
    for i in range(n):
        forward(100)
        dot()
        left(360/n)

done()
```

`polygon2.py`

5.6 Dessiner une étoile

Dessiner une étoile est similaire à dessiner un polygone régulier. En fait, on peut même considérer une étoile comme un polygone régulier. Pour une étoile, on a aussi ces deux conditions qui sont valables:

- touts les côtés ont la même longueur
- touts les angles ont la même valeur

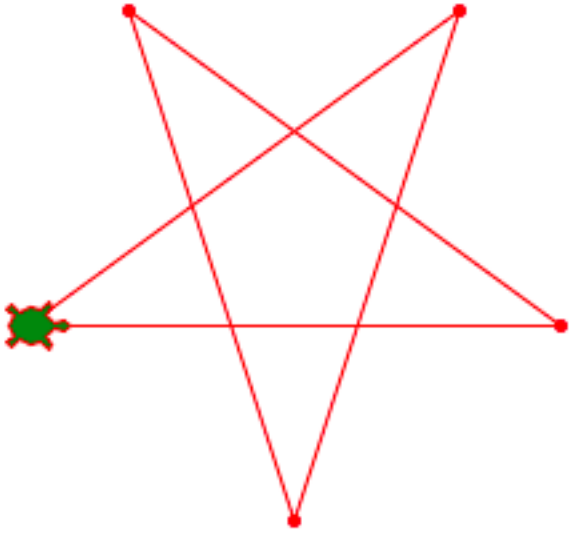
Il suffit de modifier la formule du calculer d'angle:

```
left(360/n*m)
```

Vous pouvez expérimenter avec différents valeurs pour:

```
a = 200
n = 5
m = 2
```

Tu peux constater que pour `m=1`, tu obtiens le polygone ordinaire, l'angle valant alors 360/n.



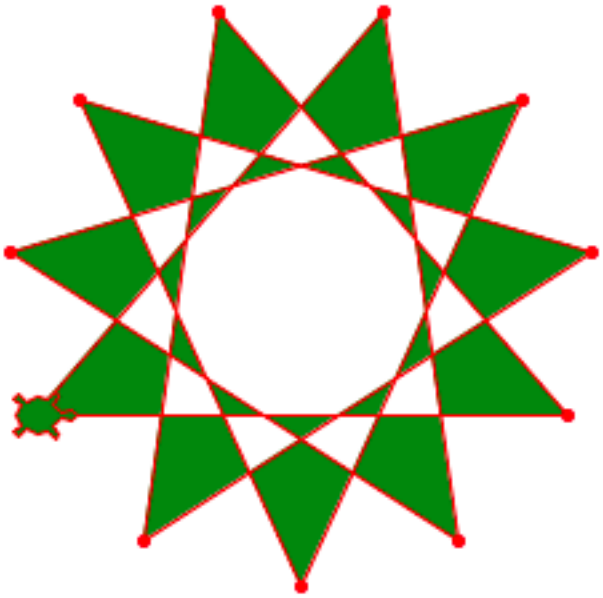
```
from turtle import *

a = 200
n = 5
m = 2

for i in range(n):
    forward(a)
    dot()
    left(360/n*m)

done()
```

`polygon3.py`

5.7 Dessiner une étoile coloriée

Pour remplir le dessin de l'étoile il suffit d'appeler ces deux fonctions avant et après les lignes de code qui créent le dessin:

```
begin_fill()
end_fill()
```



```
from turtle import *

a = 200
n = 11
m = 4

begin_fill()
for i in range(n):
    forward(a)
    dot()
    left(360/n*m)
end_fill()

done()
```

polygon4.py

5.8 Dessiner un arc en ciel

Pour dessiner un arc-en-ciel, nous définissons d'abord une liste de couleurs:

```
colors = ('red', 'orange', 'yellow', 'lightgreen', 'lightblue', 'violet')
```

Cette liste contient 6 éléments, et la **boucle for** avec une liste permet de répéter 6 fois, avec la variable `color` prenant successivement les valeurs dans la liste `colors`:

```
for color in colors:
    pencolor(color)
```

Nous commençons à dessiner un demi-cercle, donc un arc de 180 degrès et un rayon inital de 40:

```
circle(r, 180)
```

Arrivé du coté gauche de l'arc nous faisons ceci:

- tourner de 90 degrés à gauche
- soulever le stylo
- avancer le diamètre de l'arc rouge (2*r = 80)
- avancer encore l'épaisseur du trait (d=10)
- redescendre le stylo
- tourner de 90 degrés
- augmenter le rayon pour le prochain arc en orange (40+10 = 50)


```
from turtle import *

d = 10
r = 60
colors = ('red', 'orange', 'yellow', 'lightgreen', 'lightblue', 'violet')

pensize(d)
left(90)

for color in colors:
    pencolor(color)
    circle(r, 180)
    left(90)
    up()
    forward(2*r + d)
    down()
    left(90)
    r += d

write('rain bow', font=(None, 36))
done()
```

rainbow.py

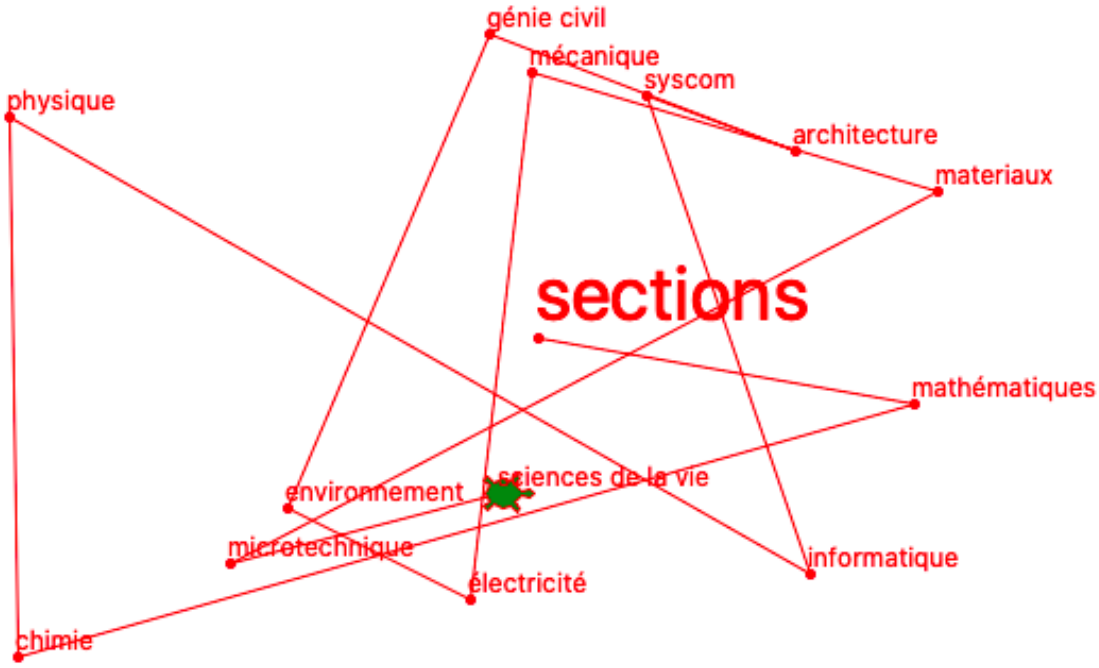
CHAPTER 6

L'aléatoire

L'aléatoire est ce qui relève du hasard. Quand on roule un dé, son résultat est aléatoire.

Dans des jeux ou des animations, il est souvent nécessaire de pouvoir calculer des valeurs aléatores. Le module `random` permet de trouver des valeurs aléatoires.

Partant du centre, la tortue visite les 13 sections de l'EPFL qui se trouvent tous à des positions aléatoires sur cette carte.



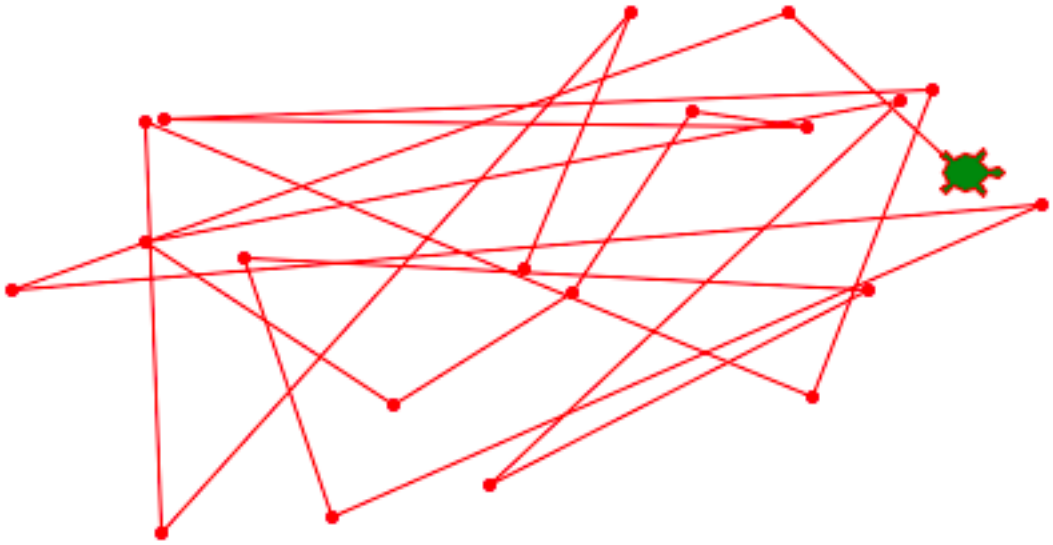
`epfl4.py`

6.1 Position aléatoire

Pour pouvoir utiliser des fonctions aléatoirs nous devons d'abord importer le module `random`:

```
import random
```

La fonction `random.randint(-100, 100)` retourne une valeur aléatoire qui se situe entre les deux valeurs -100 et 100. Pour obtenir une position aléatoire, nous devons calculer deux valeurs, x et y.



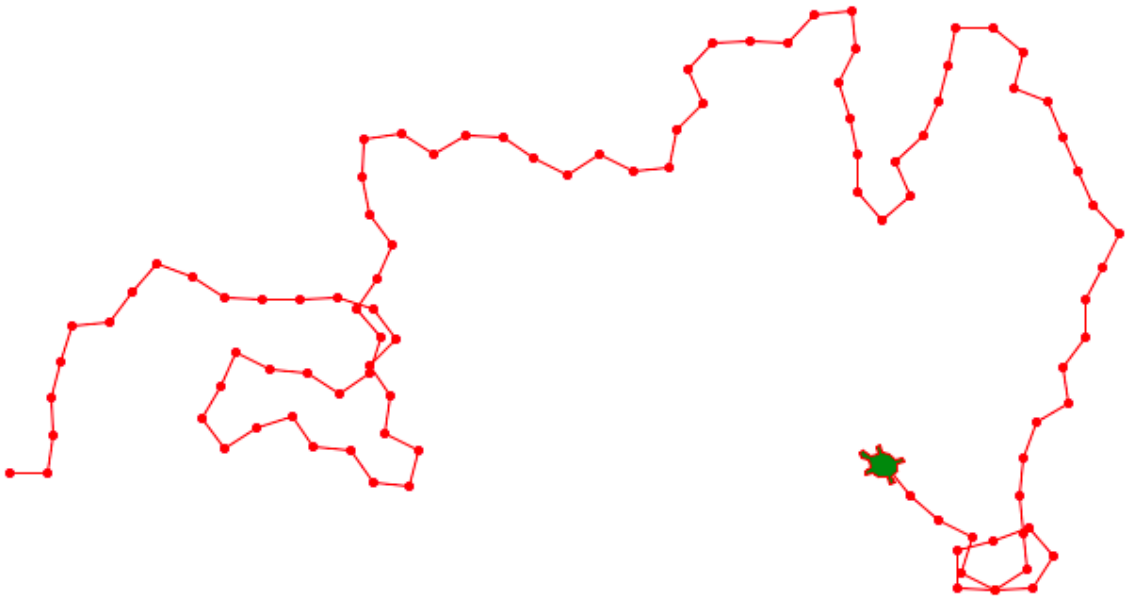
```
from turtle import *
import random

dot()
for i in range(20):
    x = random.randint(-200, 200)
    y = random.randint(-100, 100)
    goto(x, y)
    dot()

done()
```

`random1.py`

6.2 Angle aléatoire

Pour simuler la marche aléatoire d'une fourmi, nous pouvons garder la distance de chaque pas constante, et choisir l'angle du changement de direction à chaque itération comme ceci:

```
angle = random.randint(-90, 90)
```



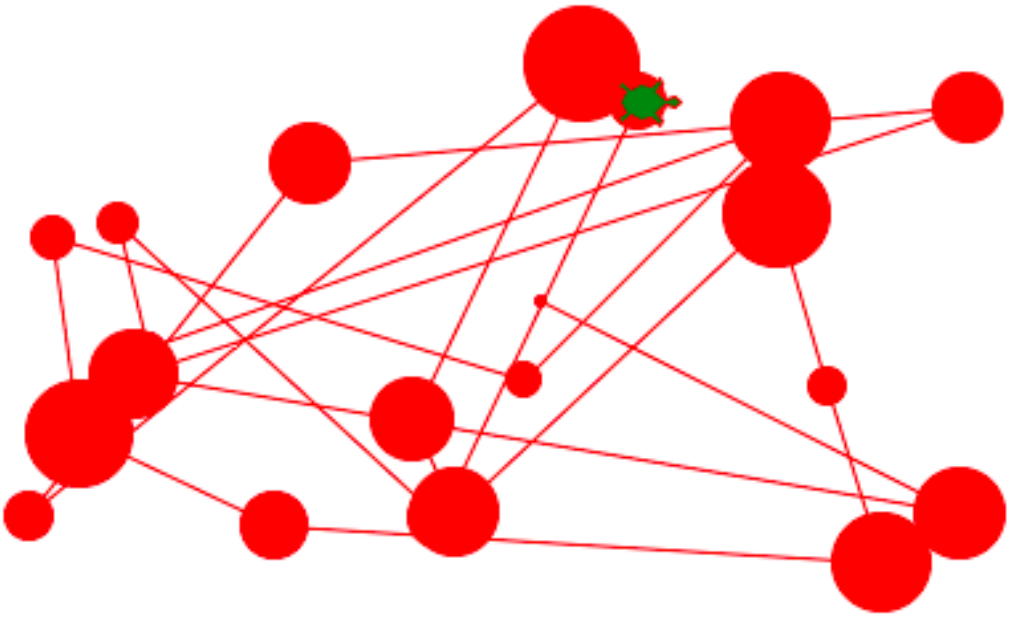
```
from turtle import *
import random

dot()
for i in range(100):
    angle = random.randint(-90, 90)
    forward(20)
    left(angle)
    dot()

done()
```

random2.py

6.3 Taille aléatoire

Ci-dessous la tortue va à une position (x, y) aléatoire et choisit une taille aléatoire dans l'intervalle [10, 50] pour dessiner un cercle:

```
size = random.randint(10, 50)
goto(x, y)
dot(size)
```



```
from turtle import *
import random

dot()
for i in range(20):
    x = random.randint(-200, 200)
    y = random.randint(-100, 100)
    size = random.randint(10, 50)
    goto(x, y)
    dot(size)

done()
```

random3.py

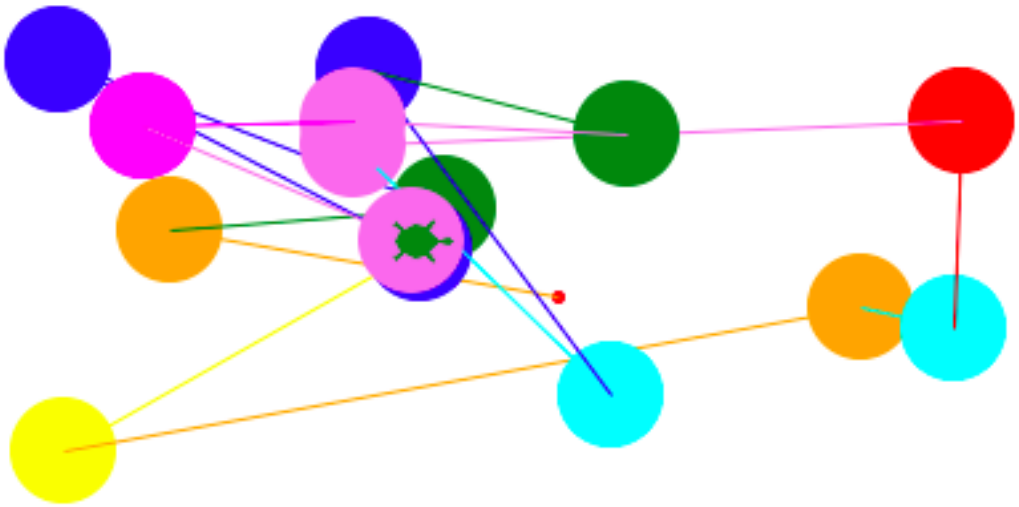
6.4 Couleur aléatoire

La fonction `random.choice(list)` permet de choisir un élément dans une liste. Il faut d'abord définir une liste:

```
colors = ('red', 'blue', 'green', 'violet', 'yellow', 'cyan', 'orange', 'magenta')
```

Ensuite un élément aléatoire est choisi dans cette liste et utilisé comme nouvelle couleur pour la tortue:

```
color = random.choice(colors)
pencolor(color)
```



```
from turtle import *
import random

colors = ('red', 'blue', 'green', 'violet', 'yellow', 'cyan',
    'orange', 'magenta')


dot()
for i in range(15):
    x = random.randint(-200, 200)
    y = random.randint(-100, 100)
    color = random.choice(colors)
    pencolor(color)
    goto(x, y)
    dot(40)

done()
```

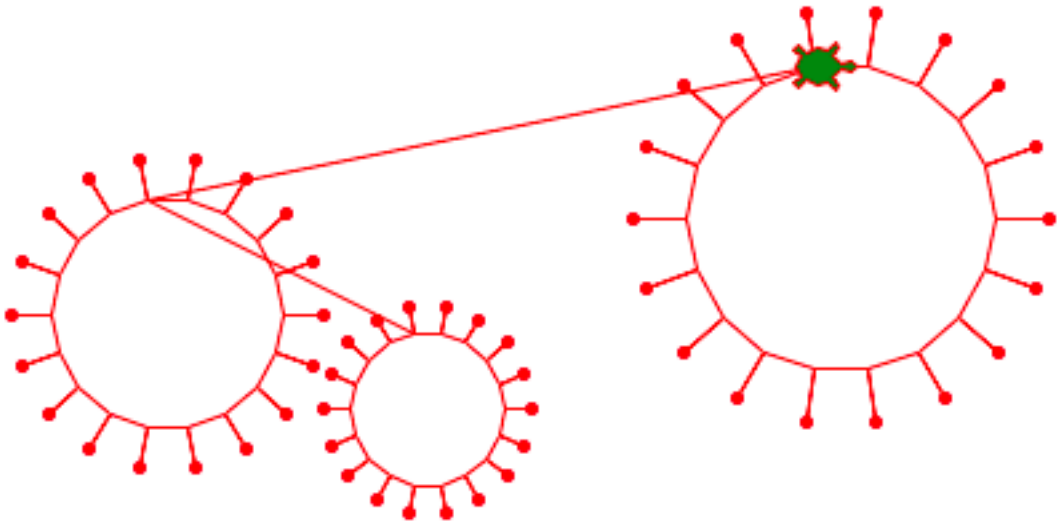
random4.py

CHAPTER 7

La fonction avec arguments

Une fonction permet de donner un nom à un bout de code. C'est similaire à une variable qui associe un nom à une valeur.

Dans l'exemple ci-dessous, la tortue dessine un virus. Une fonction `virus(d)` a été définie qui permet à la tortue de dessiner des virus de tailles différentes. La fonction a un paramètre `d` qui est la taille du la projection du virus.



`virus.py`

7.1 Définir une fonction

Nous allons retourner à plusieurs figures que nous avons faites dans les chapitres précédents et nous allons les refaire à l'aide de fonctions.

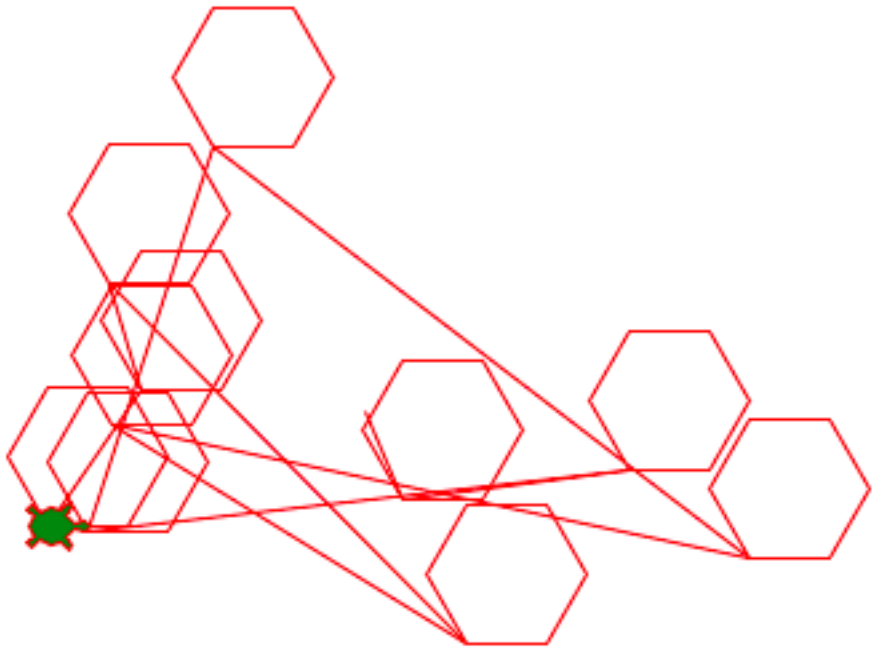
Nous définissions la fonction `polygon` avec les lignes:

```
def polygon():
    for i in range(6):
        forward(30)
        left(60)
```

Cette fonction dessine un hexagone ayant une longueur de côté de 30 pixels. Dans la suite nous appelons cette fonctions 10 fois avec tout simplement:

```
polygon()
```

Dans la boucle nous choisissons 10 fois une position aléatoire ou dessiner l'hexagone.



```
from turtle import *
import random

def hexagon():
    for i in range(6):
        forward(30)
        left(60)

for i in range(10):
    x = random.randint(-200, 200)
    y = random.randint(-100, 100)
    goto(x, y)
    hexagon()
```

(continues on next page)

(continued from previous page)

```
done()
```

function1.py

7.2 Une fonction avec des arguments

La fonction `hexagon()` utilisée définis dans la section précédente est certes pratiques, mais pas très flexible. Elle ne permet que de dessiner des hexagones, et seulement d'une taille fixe.

Pour avoir cette flexibilité, on doit donner des arguments à la fonction. Les arguments d'une fonction sont des variables qui sont indiquées dans les parenthèses lors de l'appel de la fonction. Ces arguments permettent de modifier la tâche de la fonction.

Nous allons donner deux arguments à notre fonction `polygon(n, a)`:

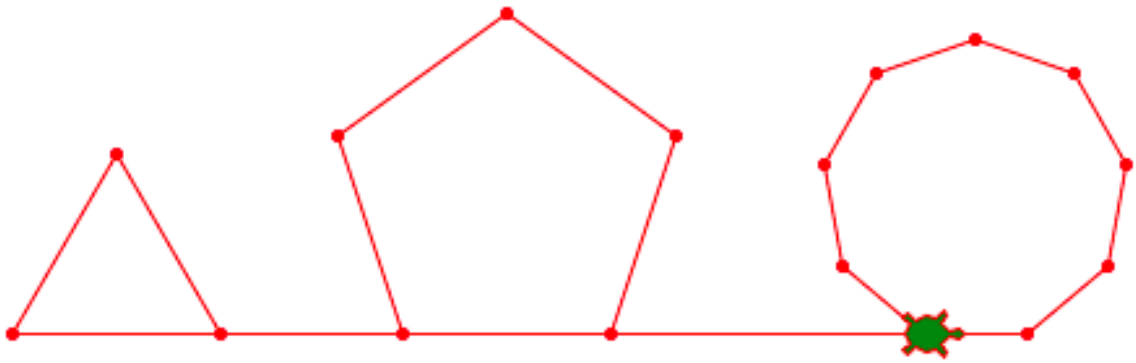
- le nombre de sommets **n**
- la longueur d'un côté **a**

Voici la nouvelle définition:

```
def polygon(n, a):
    for i in range(n):
        forward(a)
        dot()
        left(360/n)
```

Nous pouvons maintenant appeler la fonction 3 fois avec des arguments différents:

```
polygon(3, 80)
polygon(5, 80)
polygon(9, 40)
```



```
from turtle import *

def polygon(n, a):
    for i in range(n):
        forward(a)
        dot()
        left(360/n)
```

(continues on next page)

(continued from previous page)

```
back(200)
polygon(3, 80)

forward(150)
polygon(5, 80)

forward(200)
polygon(9, 40)

done()
```

function2.py

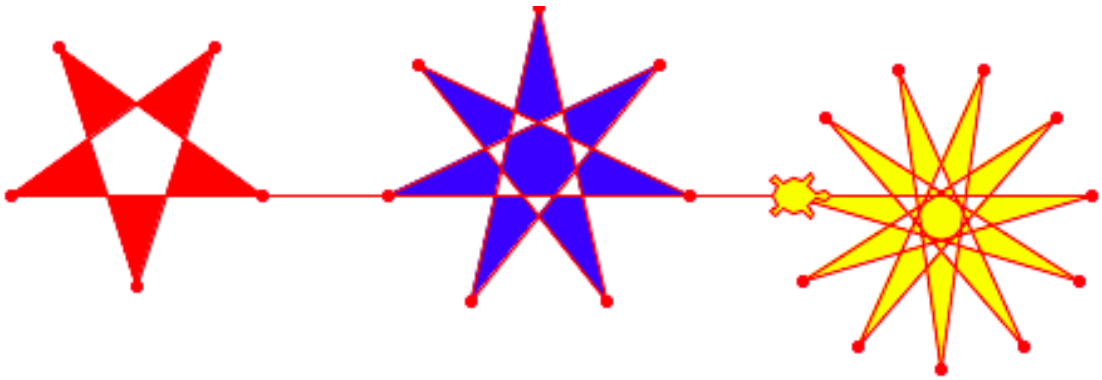
7.3 Une fonction avec 4 arguments

Nous reprenons l'étoile vu précédemment. Nous allons la transformer en fonction avec 4 arguments:

- le nombre de sommets **n**
- la distance vers le prochain sommet **m**
- la longueur d'un côté **a**
- la couleur de remplissage **color**

On appelle maintenant la fonction en indiquant 4 arguments:

```
star(5, 2, 100, 'red')
star(7, 3, 120, 'blue')
star(11, 6, 120, 'yellow')
```



```
from turtle import *

def star(n, m, a, color):
    fillcolor(color)
    begin_fill()
    for i in range(n):
        forward(a)
        dot()
        left(360/n*m)
```

(continues on next page)

(continued from previous page)

```
    end_fill()

back(200)
star(5, 2, 100, 'red')

forward(150)
star(7, 3, 120, 'blue')

forward(160)
star(11, 6, 120, 'yellow')

done()
```

function3.py

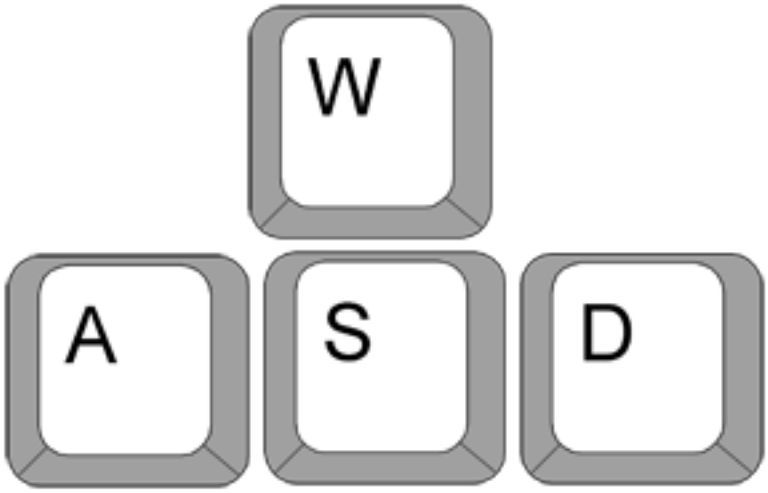
CHAPTER 8

Evénements

Quand tu utilises le clavier ou la souris de ton ordinateur, tu t'y attends qu'il réagit immédiatement. En informatique on appelle un telle action de l'utilisateur un **event**. Un programme est toujours à l'écoute des événements et répond par des actions.

8.1 Utiliser les touches

Beaucoup de jeux utilisent des touches pour faire bouger le personnage du jeu. Souvent ce sont les touches WASD du clavier.



Tu peux programmer ta tortue pour la bouger avec ces touches. La fonction `onkey(move_forward, 'w')` associe la touche **W** avec la fonction `move_forward`.

Tu dois définir cette fonction, par exemple avancer de 30 pixels:

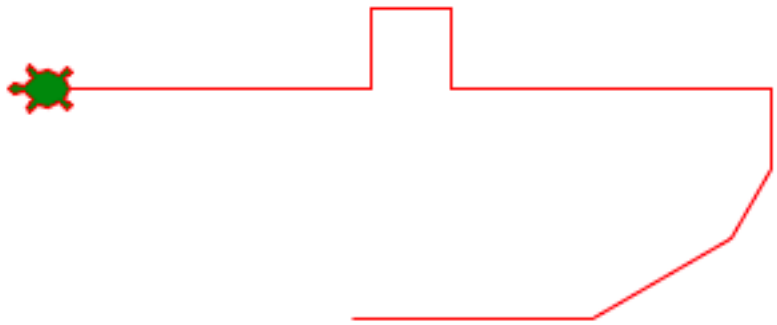
```
def move_forward():
    forward(30)
```

On appelle ça de définir une fonction de **callback** pour l'événement. Voici la définition des 4 fonctions de callback pour les 4 touches de direction WASD:

```
onkey(move_forward, 'w')
onkey(move_backward, 's')
onkey(turn_left, 'a')
onkey(turn_right, 'd')
```

Pour que la tortue commence à écouter aux événements, tu dois appeler `listen()`.

Essaye maintenant de contrôler la tortue avec les touches WASD et de faire un dessin.

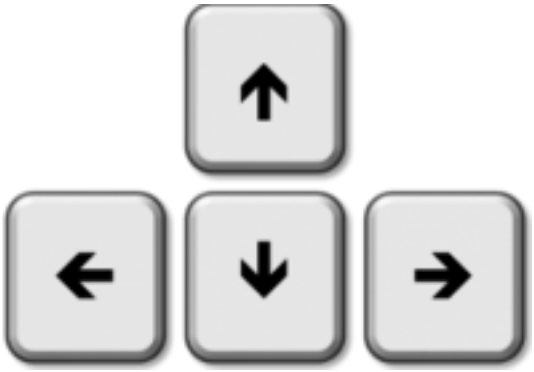


```
from turtle import *

def move_forward():
    forward(30)

def move_backward():
    backward(30)

def turn_left():
    left(30)

def turn_right():
    right(30)

onkey(move_forward, 'w')
onkey(move_backward, 's')
onkey(turn_left, 'a')
onkey(turn_right, 'd')
listen()

forward(0)
done()
```

`event1.py`

8.2 Utiliser les flèches

Si tu préfère les flèches, tu peux aussi programmer ta tortue pour la bouger avec les touches de direction.



Les 4 touches de direction ont les noms **Up, Down, Left, Right**. Voici la défintion des 4 fonctions callback:

```
onkey(move_forward, 'Up')
onkey(move_backward, 'Down')
onkey(turn_left, 'Left')
onkey(turn_right, 'Right')
```

Tu peux combiner les touche WASD avec les flèches. Ceci te donne deux méthodes différentes pour bouger la tortue.

```
from turtle import *

def move_forward():
    forward(30)

def move_backward():
    backward(30)

def turn_left():
    left(30)

def turn_right():
    right(30)

onkey(move_forward, 'Up')
onkey(move_backward, 'Down')
onkey(turn_left, 'Left')
onkey(turn_right, 'Right')
listen()

forward(0)
done()
```

`event2.py`

8.3 Effacer l'écran

Tu peux ajouter d'autres fonctions, comme:

- effacer l'écran (clear)

- retourner la tortue à la position initiale (home)
- remettre en situation initial (reset)
- quitter (bye)

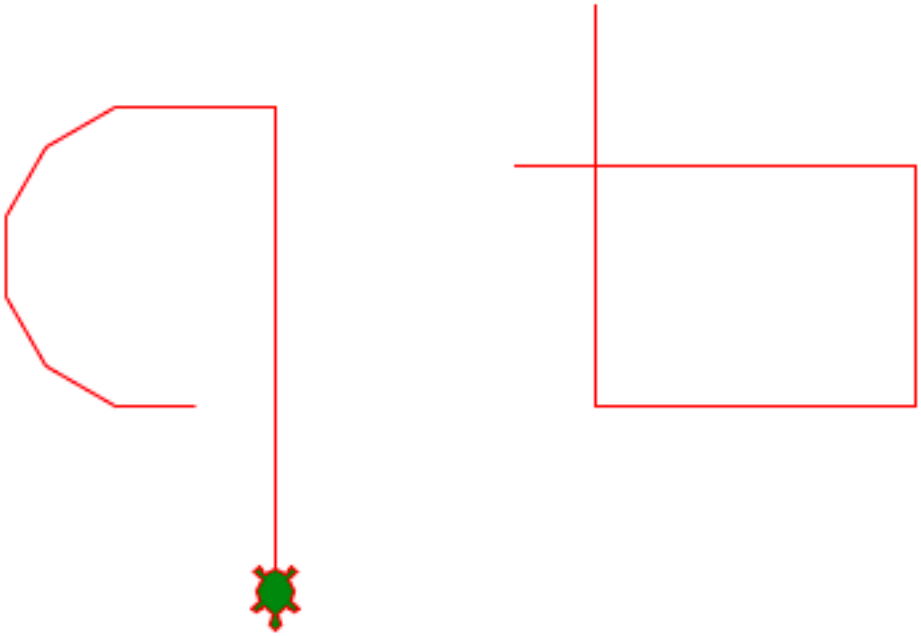
Comme ces fonctions n'ont pas d'argument, tu n'as pas besoin de redéfinir séparément ces fonctions. Tu peux directement mettre leur nom (sans les parenthèses) dans la définition des fonctions callback:

```
onkey(home, 'h')
onkey(clear, 'c')
onkey(reset, 'r')
onkey(bye, 'q')
```

Tu peux également ajouter des raccourcis pour faire monter et descendre le stylo:

```
onkey(up, 'u')
onkey(down, 'd')
```

Ceci te permet de dessiner des lignes séparées.



Voici le programme en entier.

```
from turtle import *

def move_forward():
    forward(30)

def move_backward():
    backward(30)

def turn_left():
    left(30)
```

(continues on next page)

(continued from previous page)

```
def turn_right():
    right(30)

onkey(move_forward, 'Up')
onkey(move_backward, 'Down')
onkey(turn_left, 'Left')
onkey(turn_right, 'Right')

onkey(home, 'h')
onkey(clear, 'c')
onkey(reset, 'r')
onkey(bye, 'q')

onkey(up, 'u')
onkey(down, 'd')

listen()

forward(0)
done()
```

event3.py

8.4 La fonction lambda

La **fonction lambda** est une courte fonction définit définit en une seul ligne. Cette fonction ne porte pas de nom, et on l'appelle aussi **fonction anonyme**. Au lieu d'écrire

```
def move_forward():
    forward(30)
```

tu peux écrire

```
lambda : forward(30)
```

Avec les fonctions lambda notre programme devient très court.

```
from turtle import *

onkey(lambda: forward(30), 'Up')
onkey(lambda: backward(30), 'Down')
onkey(lambda: left(30), 'Left')
onkey(lambda: right(30), 'Right')
listen()

forward(0)
done()
```

event4.py

8.5 Contrôler deux tortues

Dans la **programmation orienté-objet** on modélise le monde par

- des **classes**,
- des **objets**, et
- des **méthodes**.

Notre tortue est un objet de la classe `Turtle`. Jusqu'à maintenant tu as toujours travaillé avec un seul objet. Mais tu peux créer plusieurs objets (tortues) et leur donner des noms différents.

Ceci définit deux objets différents: une tortue **Alice** et une tortue **Bob**:

```
alice = Turtle()
bob = Turtle()
```

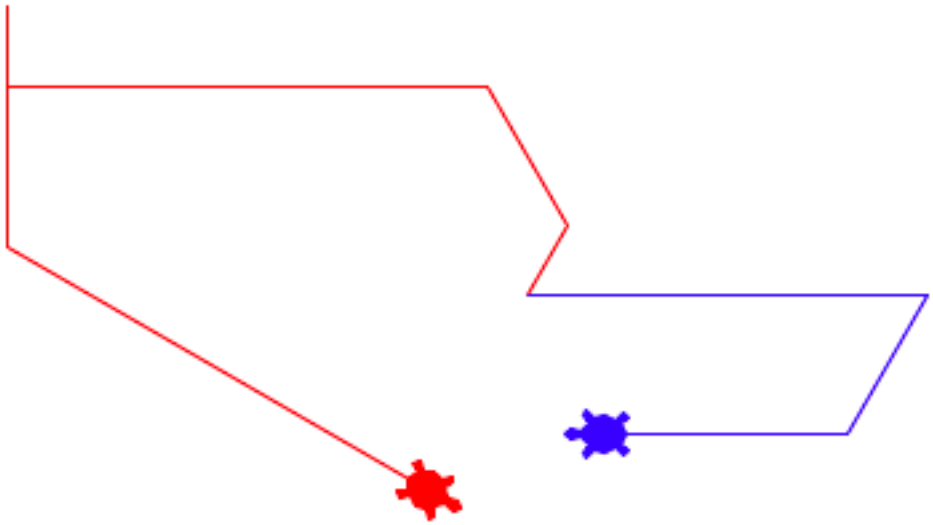
Un objet a des **attributs** tel que sa couleur. Tu peux donner une couleur différente à Alice et à Bob:

```
alice.color('red')
bob.color('blue')
```

Tu as déjà vu les méthodes (forward, left, right, etc.) que tu as utilisé pour l'objet par défaut `turtle`. Pour contrôler une tortue spécifique tu ajoute la méthode à son nom. Par exemple:

```
alice.forward(30)
bob.left(30)
```

Tu peux utiliser les touches WASD pour contrôler le mouvement d'Alice et les touches de direction pour contrôler le mouvement de Bob.



```
from turtle import *

alice = Turtle()
bob = Turtle()

alice.color('red')
bob.color('blue')

onkey(lambda: alice.forward(30), 'w')
onkey(lambda: alice.backward(30), 's')
onkey(lambda: alice.left(30), 'a')
```

(continues on next page)

(continued from previous page)

```
onkey(lambda: alice.right(30), 'd')

onkey(lambda: bob.forward(30), 'Up')
onkey(lambda: bob.backward(30), 'Down')
onkey(lambda: bob.left(30), 'Left')
onkey(lambda: bob.right(30), 'Right')

listen()
done()
```

`event5.py`

8.6 Coordonnées absolues

Les commandes *left, right, forward, backward* se réfèrent à la position et l'orientation actuelle de la tortue. Quand la tortue est tourné vers le bas, les directions pour nous sont invérsés, et c'est difficile de la controler.

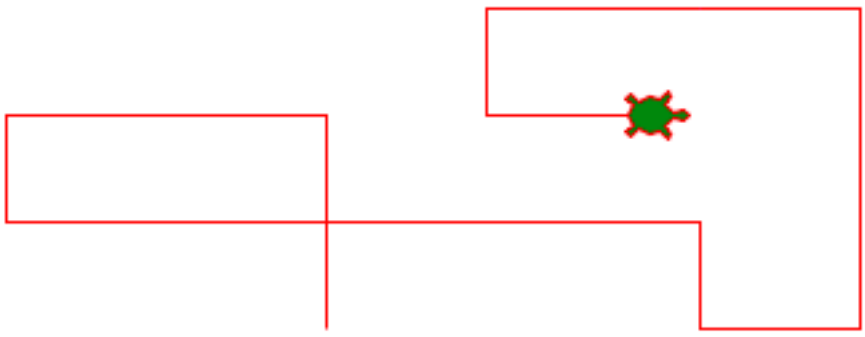
Une autre façon de controler la tortue est d'utiliser des coordonnées (x, y). Nous les définissons au début du programme, ainsi qu'une distance de déplacement `d`

```
x = 0
y = 0
d = 20
```

A l'intérieur des 4 fonctions nous voulons accéder à ces variables globales. Pour ne pas créer une nouvelle variable locale de même nom, nous devons declarer les valeurs que nous affectons avec le mot-clés `global`:

```
global x
global y
```

Avec la touche **Up/Down** nous incrémentons ou décrémentons la variable y. Avec la touche **Left/Right** nous incrémentons ou décrémentons la variable x.



```
from turtle import *

x = 0
y = 0
d = 20
```

(continues on next page)

(continued from previous page)

```
def up():
    global y
    y = y + d
    setposition(x, y)

def down():
    global y
    y = y - d
    setposition(x, y)

def left():
    global x
    x = x - d
    setposition(x, y)

def right():
    global x
    x = x + d
    setposition(x, y)

onkey(up, 'Up')
onkey(down, 'Down')
onkey(left, 'Left')
onkey(right, 'Right')

listen()
forward(0)
done()
```

event6.py

8.7 Allers vers la souris

Tu peux aussi reagir au clic de la souris. Utilise la fonction

```
onscreenclick(move)
```

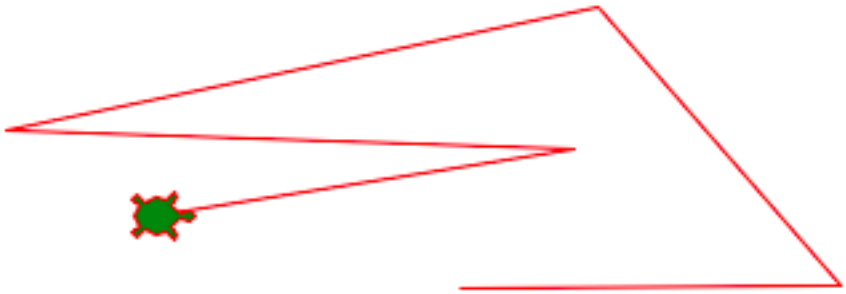
pour désigner la fonction `move` comme fonction de callback pour le clic de la souris.

```
def move(x, y):
    setposition(x, y)
```

Tu pourrais encore raccourcir ceci et directement déclarer:

```
onscreenclick(setposition)
```

Toutes les lignes sont des segments droit, et la tortue ne change pas d'orientation.



```
from turtle import *

def move(x, y):
    setposition(x, y)

onscreenclick(move)
listen()

forward(0)
done()
```

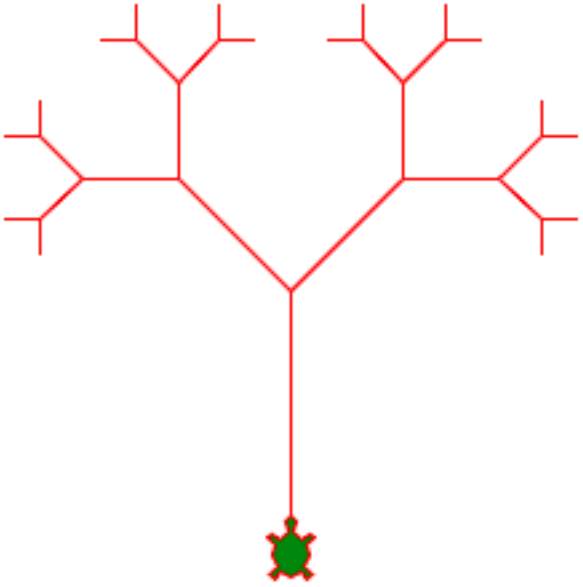
event7.py

CHAPTER 9

Récursivité

En informatique une fonction qui contient un appel à elle-même est appelé récursif.

9.1 Un arbre récursif



```
from turtle import *

d = 0.6          # decreasing factor
alpha = 45       # turning angle
```

(continues on next page)

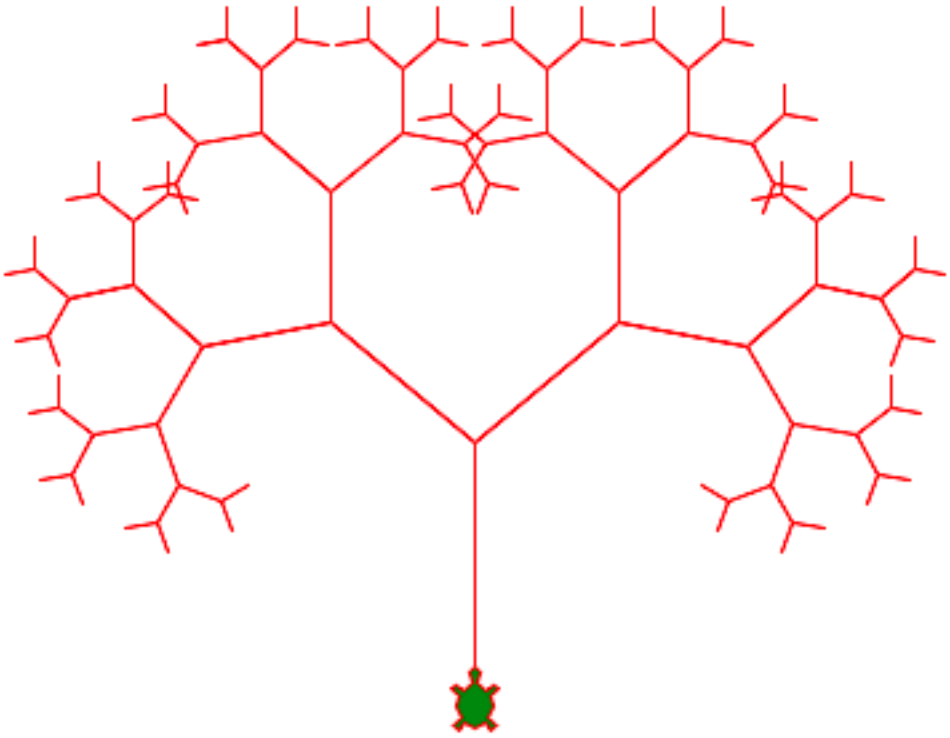
(continued from previous page)

```
l = 100          # initial length

def tree(a):
    if a < 10:
        return
    else:
        forward(a)
        left(alpha)
        tree(a * d)
        right(2 * alpha)
        tree(a * d)
        left(alpha)
        backward(a)

left(90)
backward(l)
tree(l)

done()
```

`tree1.py`

En changeant les paramètres:

```
d = 0.7          # decreasing factor
alpha = 50       # turning angle
l = 100          # initial length
```



CHAPTER 10

Indices and tables

- genindex
- modindex
- search