Seven Ways to Use Python‘s New Turtle Module

-1. Preliminary Remarks

-1.1

-1.1 About these slides

- This talk will essentially consist in an interactive session with IDLE using the turtle module.
- Additionally, I'll show and explain a selection of demo scripts which use the module's main features

- So these slides will not show exactly what you will see during the talk, but can only serve as an overview and summing-up of what you will hear and see.

- Consequently, these slides will not (or at best marginally) be used during the presentation.

-1. Preliminary Remarks

-1.2 Why seven ways?

-1.2

- The turtle module was conceived primarily as a tool for teaching programming in a way that allows for using a range of different programming styles as well as different approaches to doing geometry.

- The turtle module‘s API consists of 7 classes and 80+ methods and functions derived from these methods. For teaching purposes, especially in the introduction phase, one certainly would prefer to use a smaller subset of functions/commands.

- So my intention is to show that you have “seven ways“ to do interesting and useful things by choosing and using only an appropriate subset of all of the module‘s features.

0. Introduction

0.1 History

0.1

- Since Python 2.6/3.0, Python has had a new turtle module.
- Its development was based entirely on the previous one.
- One of the requirements was compatibility with that previous module.
- Another one was to retain (and improve) the old module's benefits, and …
- … at the same time to enhance its versatility by pragmatically adding features that support different approaches to doing and using graphics.

0. Introduction

0.2 Configuration

0.2

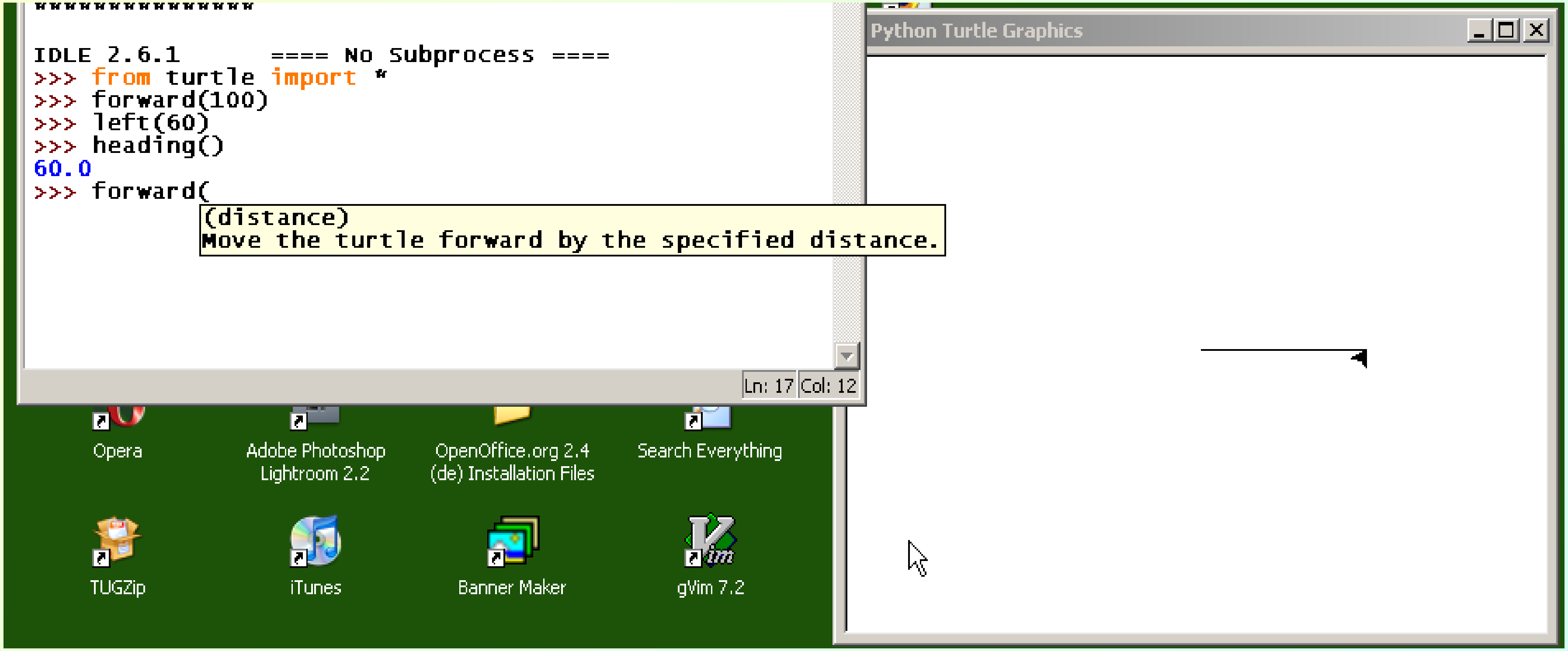
There are a few settings configurable via the turtle.cfg file, among them:

- Size and position of the turtle graphics window
- Use of a scrolled canvas
- Set the initial shape of the turtle
- Set the turtle graphics mode to “standard“ (compatible to previous turtle module) or to “logo“ (compatible with many Logo turtle graphics implementations)
- Set “help“ (and consequently tooltip-) language. The module has a helper function to produce a template for translations of the doc-strings into different languages. (Especially intended for classroom use.)

0. Introduction

0.3

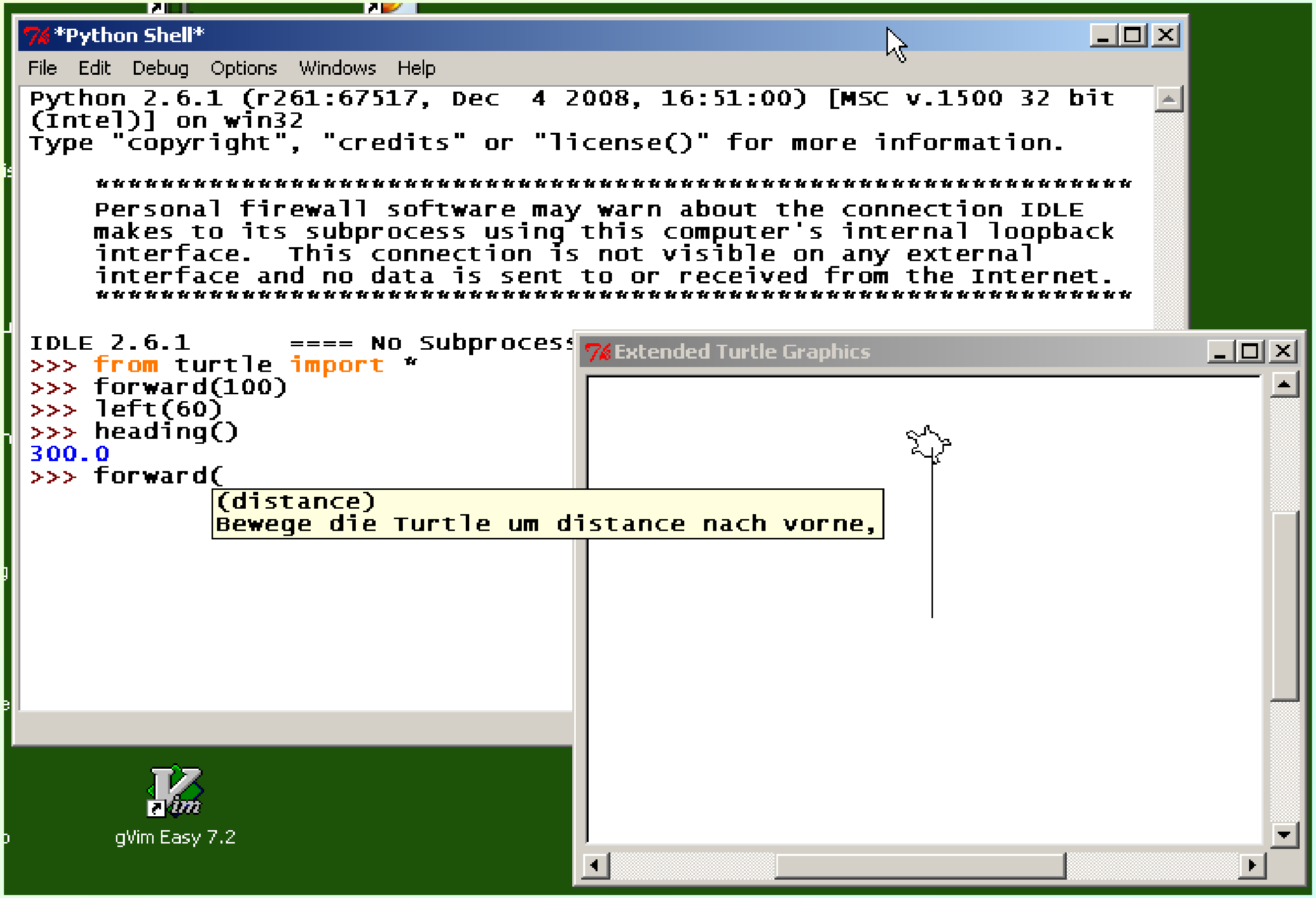
0.3 Configuration example: classic mode



0. Introduction

0.4

0.4 Configuration example: logo mode



In turtle.cfg:

```
width = 400
height = 300
canvwidth = 800
canvheight = 600
shape = turtle
mode = logo
language = german
```

1. Turtle graphics for beginners (procedural style)

1.1

1.1 Classical turtle graphics

… characterized by use of a local polar coordinate system consisting of a ruler and a goniometer, tied to an electronic animal, the turtle.

Classical commands:

```
- forward    |  fd                  back   |  bk
- right      |  rt                  left   |  lt
- penup     |  pu  |  up            pendown  |  pd
- begin_fill                        end_fill
- showturtle  |  st                 hideturtle  | ht
- pencolor,
- fillcolor
- color
- pensize
- circle
```

Special feature: `undo`

1. Turtle graphics for beginners (procedural style)

1.2

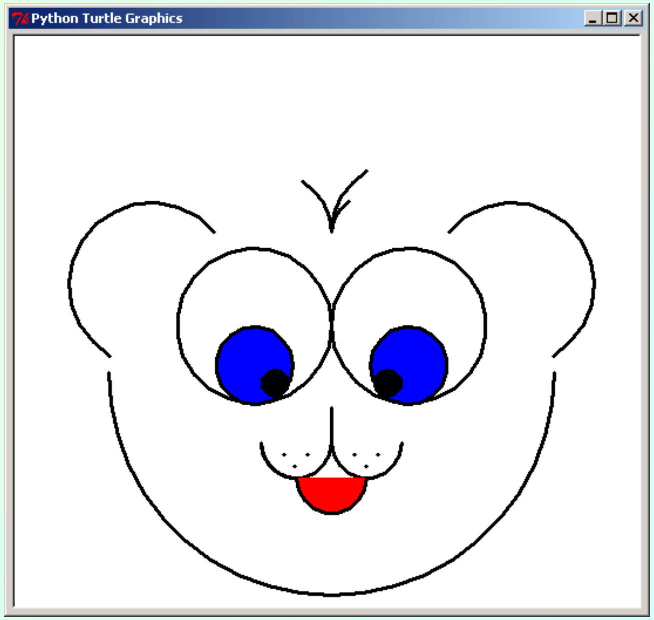
1.2 Animated drawing

… allows for visual feedback to the programmer about her ideas and programs:

- control animation (marching and turning) of the turtle via the speed command.
- Turn animation on and off using the tracer command

Sent by a reader of „Python für Kids:“

```
pensize(3)
circle(66)
rt(180)
circle(66)
pu()
lt(90)
fd(33)
rt(90)
fd(34)
pd()
fillcolor("blue")
begin_fill()
circle(33)
end_fill()
pu()
...
```



1. Turtle graphics for beginners

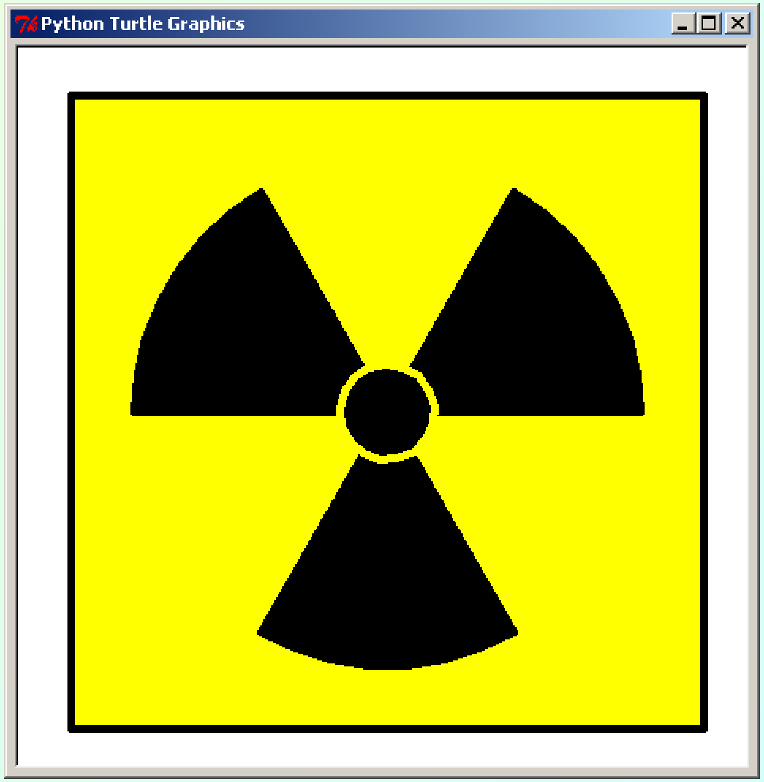
1.3 Structuring programs using functions

1.3

The animation of the drawing turtle provides feedback to trace and check the correctness of the programmer's ideas:

Elementary example: danger label "Radioactivity"

```
def square(length):
    for i in range(4):
        forward(length)
        left(90)

def sector(radius, angle):
    forward(radius)
    left(90)
    circle(radius, angle)
    left(90)
    forward(radius)
    left(180-angle)

def move(x, y):
    up()
    forward(x)
    left(90)
    forward(y)
    right(90)
    down()
```



1. Turtle graphics for beginners

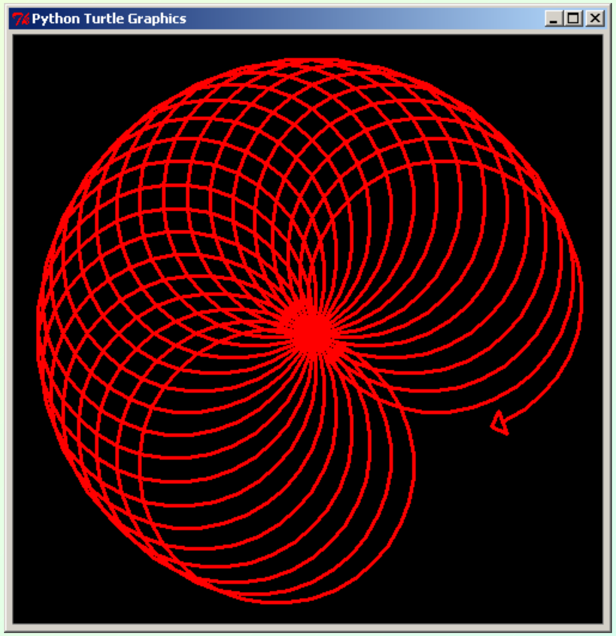
1.4

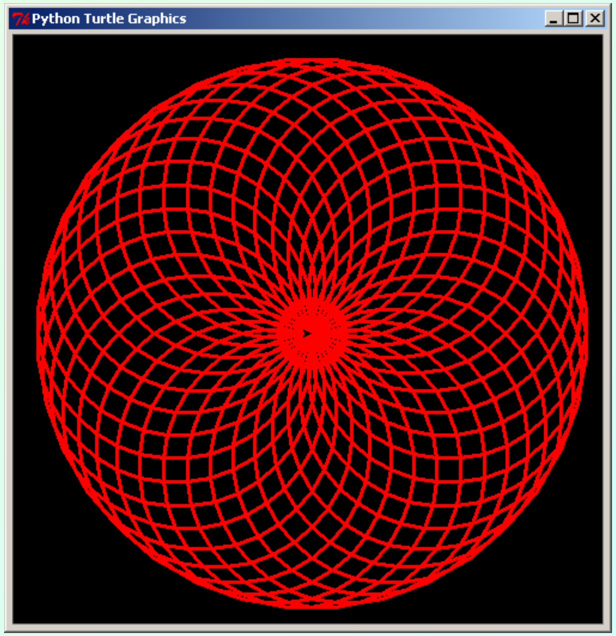
1.4 Structuring programs using functions (contd.)

The following script is modeled after an example taken from the Wikipedia article on turtle graphics:

http://en.wikipedia.org/wiki/Image:Remi_turtlegrafik.png

```
from turtle import *

def n_eck(n, size):
    for i in range(n):
        rt(360./n)
        fd(size)

def mn_eck(n, size):
    for i in range(n):
        rt(360./n)
        n_eck(n, size)

bgcolor("black")
pencolor("red")
pensize(3)
mn_eck(36, 20)
```





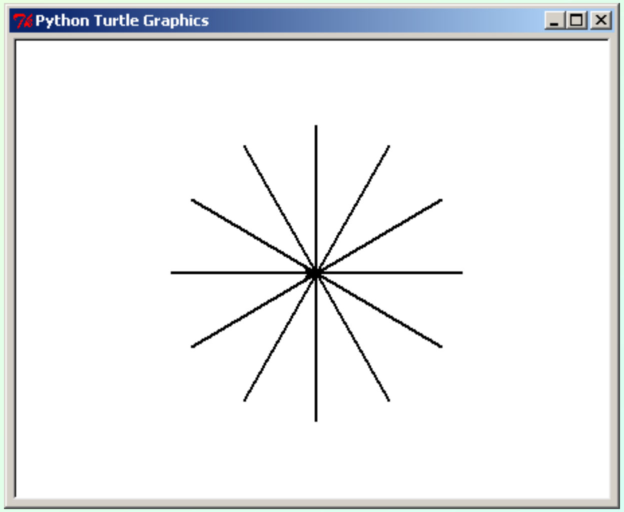
2. Cartesian geometry

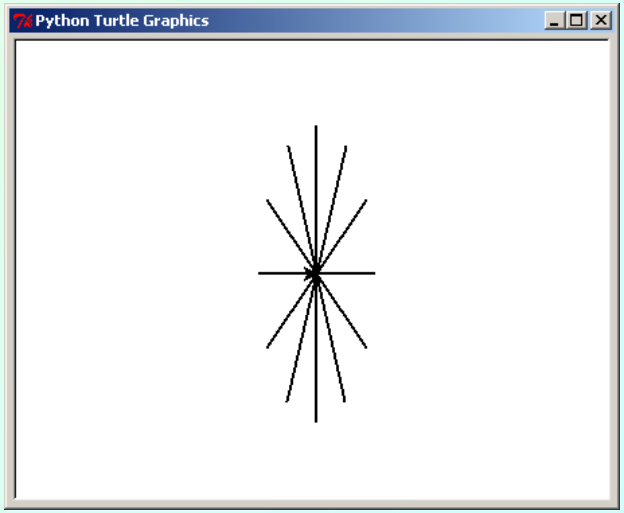
2.1 Commands related to Cartesian coordinates

```
goto(x,y)
pos()
setheading()
heading()
setworldcoordinates(xmin, ymin, xmax, ymax)
```

Depending on the ratio of x-range to y-range, angles appear distorted, which makes turtle-graphics actions appear a bit strange.

```
>>> setworldcoordinates(-2, -1.5, 2, 1.5)
>>> mode()
'world'
>>> for i in range(12):
        fd(1)
        bk(1)
        lt(30)

>>> setworldcoordinates(-5, -1.5, 5, 1.5)
```



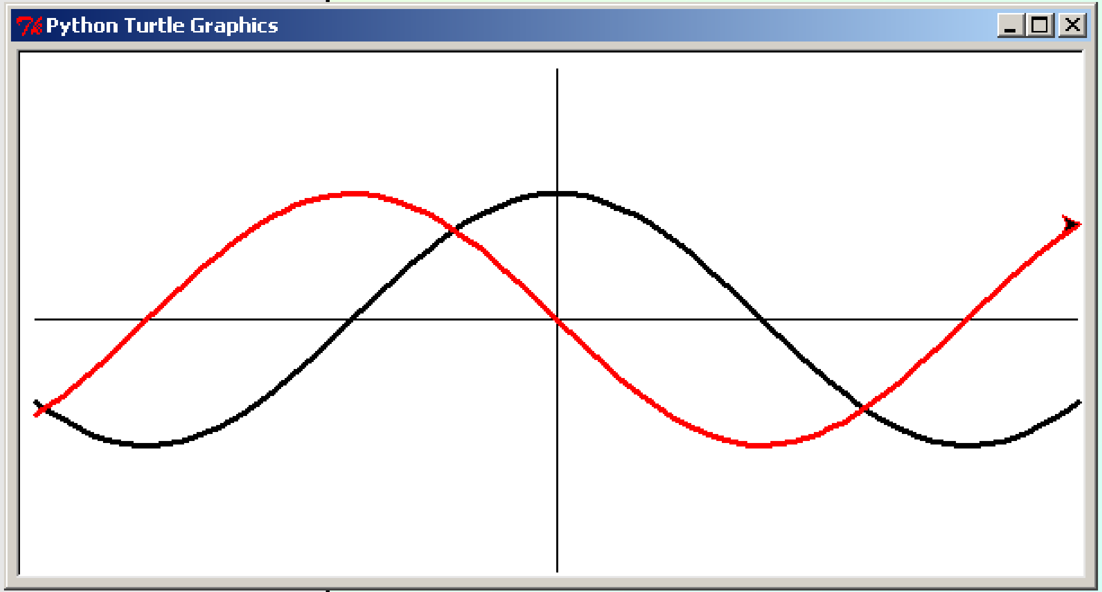


2. Cartesian geometry

2.2 Example: graphing cosine and its derivative

2.2

```
>>> def line(x1, y1, x2, y2):
        pu(); goto(x1, y1); pd(); goto(x2, y2)

>>> def graph(f):
        x = -4
        pu(); goto(x, f(x)); pd()
        while x < 4:
            x += 0.1
            goto(x, f(x))

>>> setworldcoordinates(-4,-2,4,2)
>>> line(-4,0,4,0)
>>> line(0,-2,0,2)
>>> pensize(3)
>>> from math import cos
>>> graph(cos)
>>> dx = 1.e-6
>>> def derivative(f):
        def d(x):
            return (f(x+dx)-f(x))/dx
        return d

>>> pencolor("red")
>>> graph(derivative(cos))
```



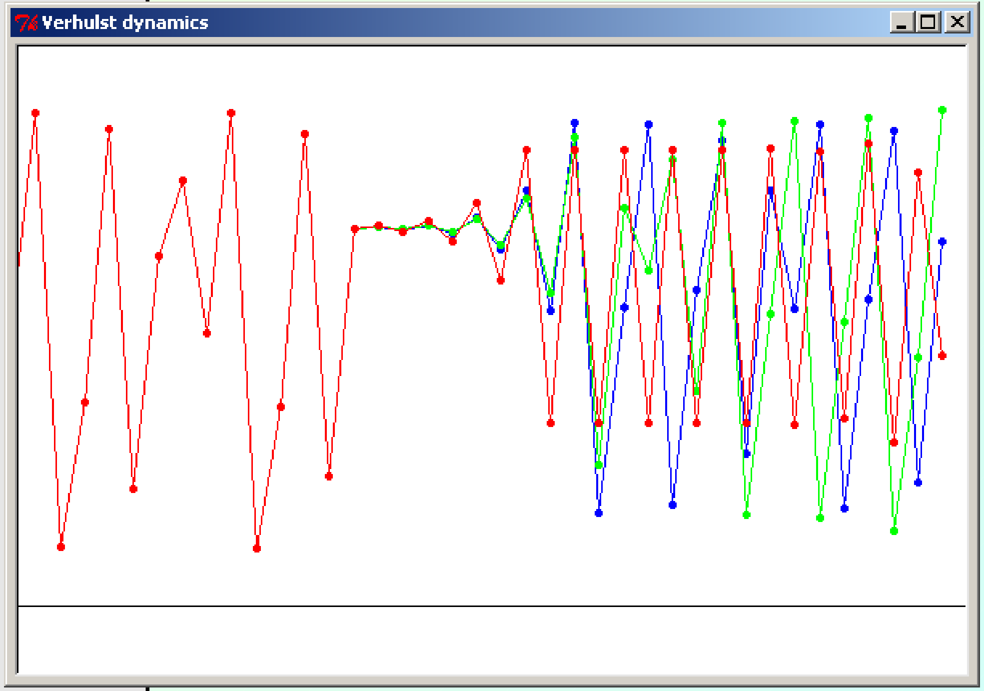
2. Cartesian geometry

2.3

2.3 Can you trust the computations of your computer?

We iterate three mathematically identical functions:

```
def f(x): return 3.9*x*(1-x)
def g(x): return 3.9*(x-x**2)
def h(x): return 3.9*x-3.9*x*x

def plot_iteration(fun,start,color):
    x = start
    pencolor(color)
    pu(); goto(0, x); pd()
    dot(5)
    for i in range(n):
        x=fun(x)
        goto(i+1,x)
        dot(5)

# … some stuff
n = 80
setworldcoordinates(-1.0,-0.1, n+1, 1.1)
plot_iteration(f, 0.35, "blue")
plot_iteration(g, 0.35, "green")
plot_iteration(h, 0.35, "red")
```



2. Cartesian geometry

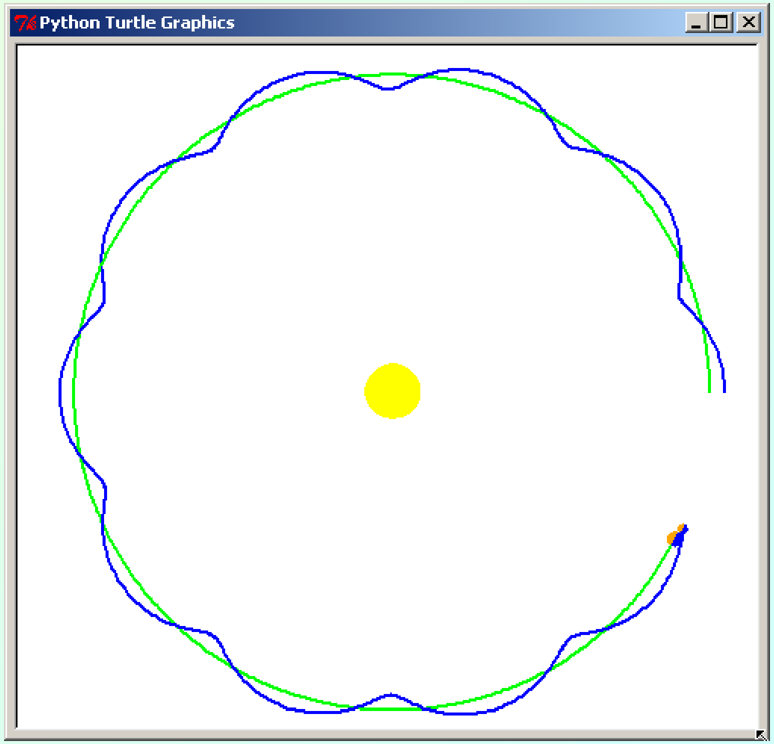
2.3 The class Vec2D

2.4

The turtle module provides and employs a 2D-vector class.
pos() returns a 2D-vector of the turtle-coordinates.

The physics-part in a simulation of a gravitational system with turtles as stars:

```
def acc(self):
    a = Vec2D(0,0)
    for planet in self.gravSys.planets:
        if planet == self: continue
        r = planet.pos()-self.pos()
        a += (G*planet.m/abs(r)**3) * r
    return a

def step(self):
    sun = self.gravSys.planets[0]
    self.setpos(self.pos() + dt*self.v)
    if self != sun:
        self.seth(self.towards(sun))
    self.a = self.acc()
    self.v = self.v + dt*self.a
```

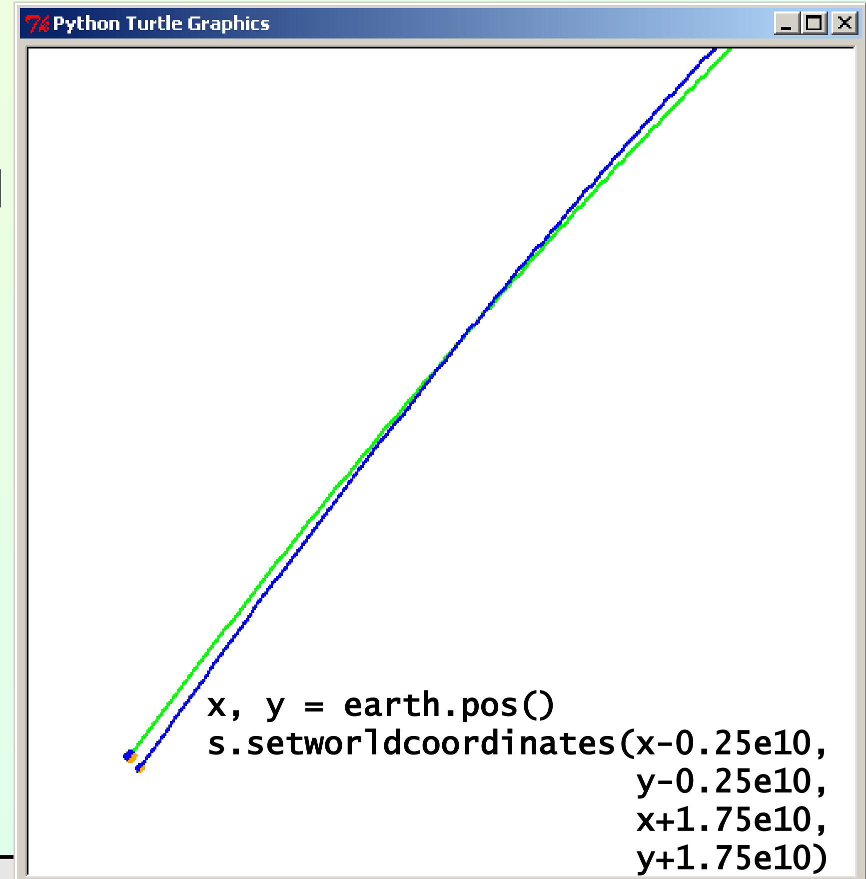


2. Cartesian geometry

2.5

2.4 A simulation of the orbit of the Earth's moon

This can be accomplished by using physical data of the solar system and **`setworldcoordinates()`.**




```
G = 6.67e-11
mS = 2.e30
mE = 6.e24
mM = 7.35e22

rE = 1.5e11
rM = 3.85e8

vE = 3.e4
vM = 1.e3
```

```
gs = GravSys()
sun = Star(mS, Vec2D(0.,0.), Vec2D(0.,0.), gs, "circle")
earth = Star(mE, Vec2D(rE,0.), Vec2D(0.,vE), gs, "planet")
moon = Star(mM, Vec2D(rE+rM,0.), Vec2D(0.,vE+vM), gs, "planet")
```

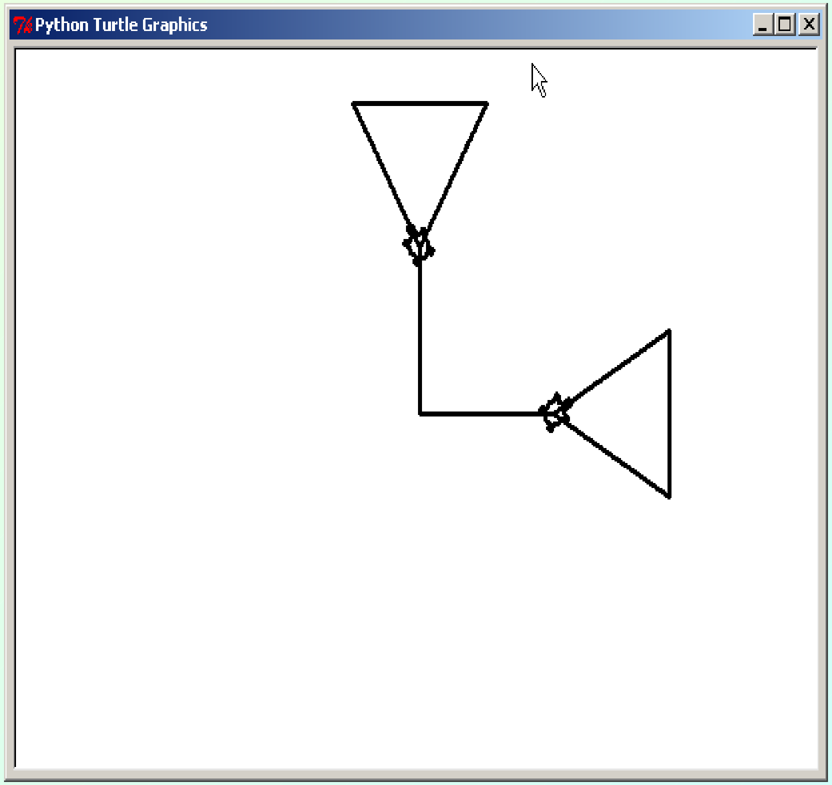
3 Turtles as objects

3.1 The classes `Screen()` and `Turtle()`

3.1

Introducing object based programming: creating turtle objects as instances of class Turtle:

```
>>> from turtle import Turtle
>>> alex = Turtle()
>>> alex.pensize(3)
>>> alex.forward(100)
>>> bert = Turtle()
>>> bert.pensize(3)
>>> bert.right(90)
>>> bert.forward(100)
>>> for turtle in alex, bert:
    turtle.left(30)

>>> for i in range(3):
        for turtle in alex, bert:
                turtle.forward(100)
                turtle.right(120)
```



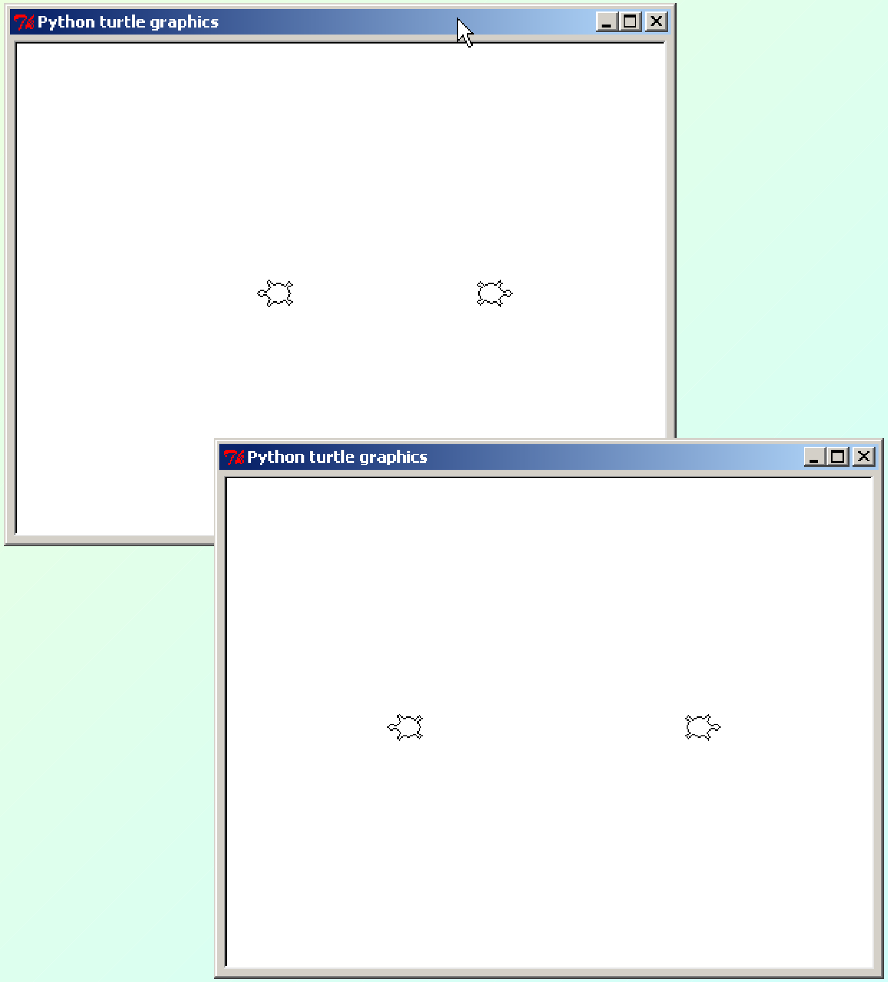
3 Turtles as objects

3.2

3.2 A gentle introduction to defining classes: how to make turtles jump

Step 1: use a function that has a turtle as first argument:

```
from turtle import Turtle
>>> fred = Turtle()
>>> def jump(turtle, distance):
    turtle.pu()
    turtle.forward(distance)
    turtle.pd()


>>> jump(fred, 100)
>>> herb = Turtle()
>>> herb.left(180)
>>> jump(herb, 100)
>>>
```



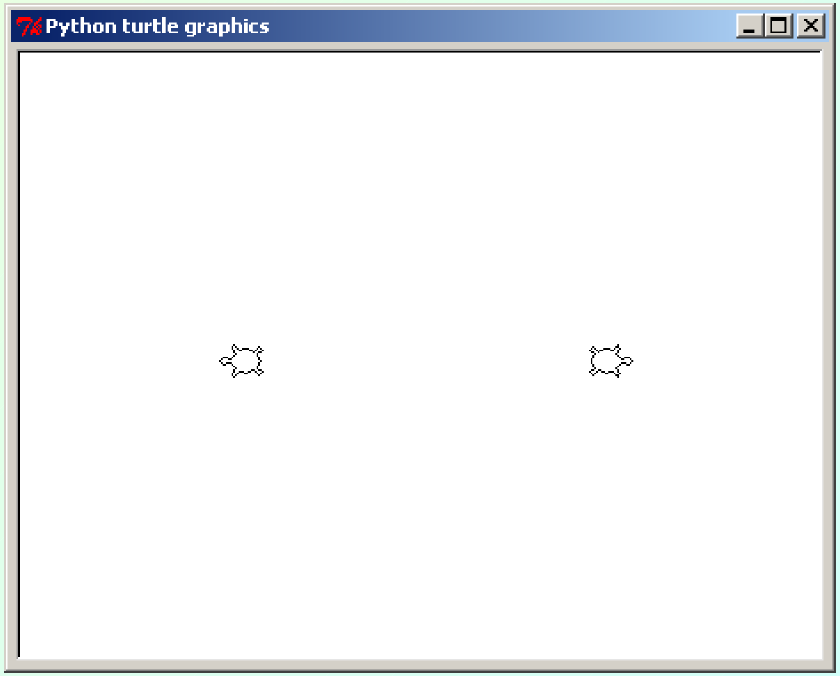
3 Turtles as objects

3.3

3.3 A gentle introduction to defining classes: how to make turtles jump (contd.)

Step 2: Bind this function to a class:

```
>>> from turtle import Turtle
>>> class MyTurtle(Turtle):
    def jump(turtle, distance):
        turtle.pu()
        turtle.forward(distance)
        turtle.pd()

>>> fred = MyTurtle()
>>> fred.jump(100)
>>> herb = MyTurtle()
>>> herb.left(180)
>>> herb.jump(100)
```



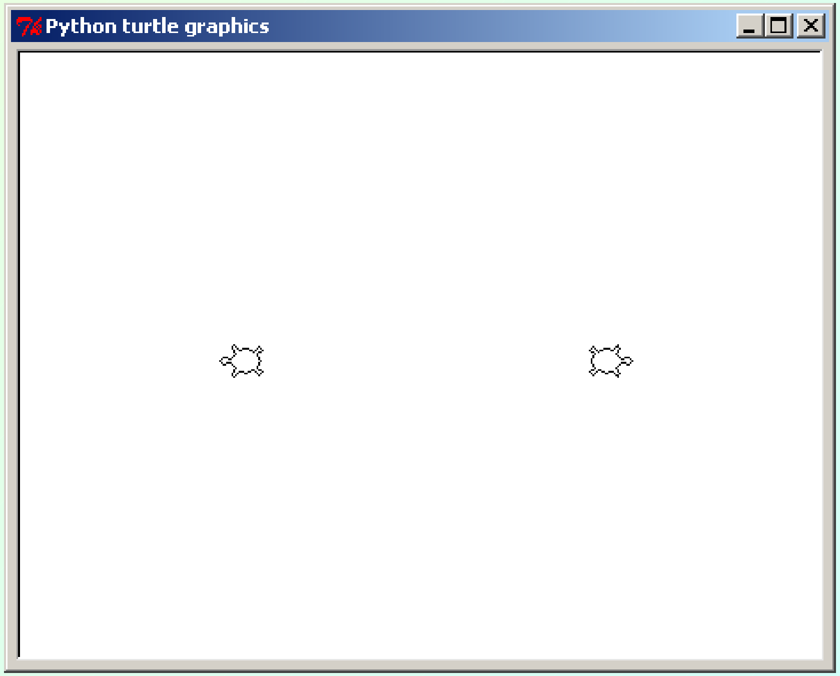
3 Turtles as objects

3.4

3.4 A gentle introduction to defining classes: how to make turtles jump (contd.)

Step 3: Observe the convention to name the first parameter **`self`**:

```
>>> from turtle import Turtle
>>> class MyTurtle(Turtle):
    def jump(self, distance):
        self.pu()
        self.forward(distance)
        self.pd()

>>> fred = MyTurtle()
>>> fred.jump(100)
>>> herb = MyTurtle()
>>> herb.left(180)
>>> herb.jump(100)
```



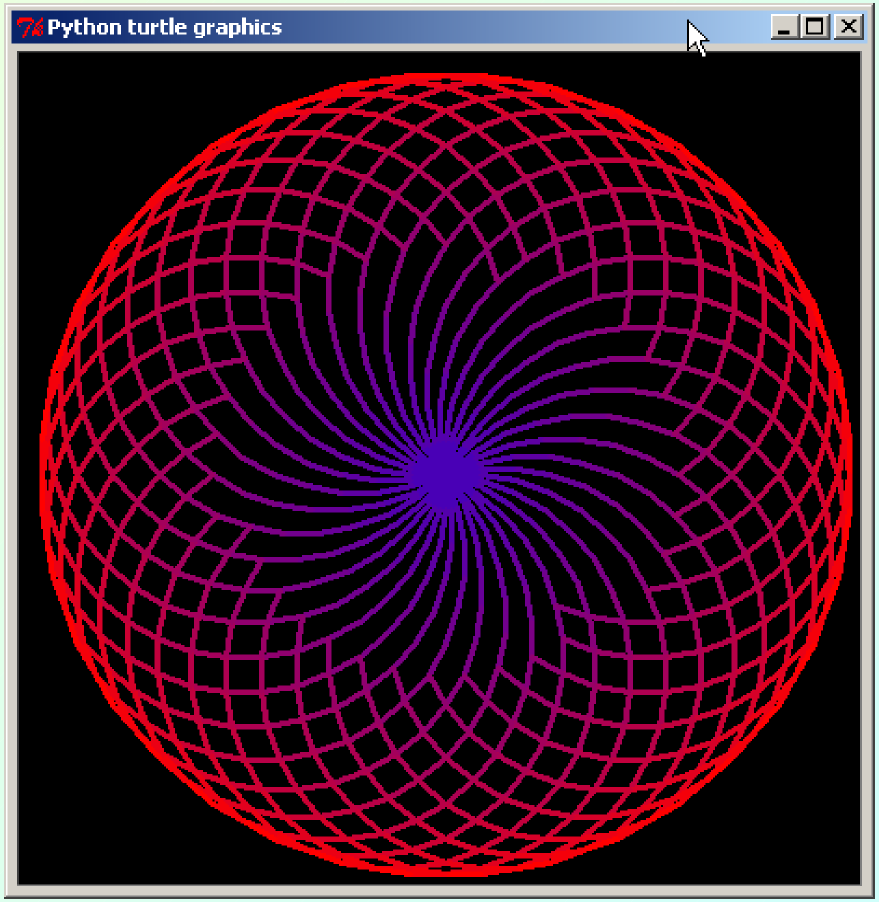
3 Turtles as objects

3.5 Wikipedia example revisited

3.5

Using multiple turtles, we can draw the Wikipedia circles in parallel:

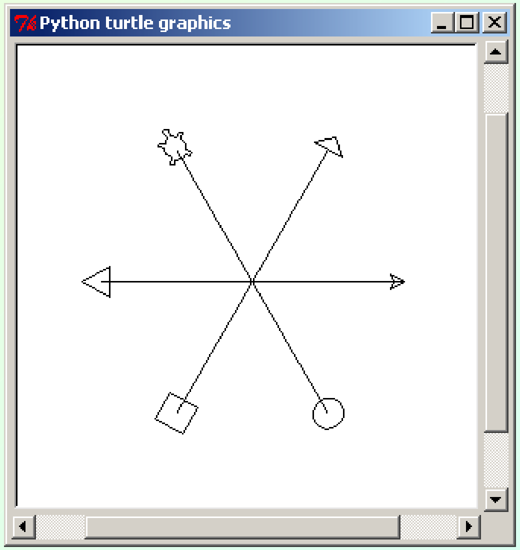
```
def create_turtles(ne):
    for i in range(ne):
        t=Turtle()
        t.ht()
        t.speed(0)
        t.seth(i*360.0/ne)
        t.pensize(3)
    return s.turtles()

def mn_eck(ne,sz):
    #create ne turtles
    myturtles = create_turtles(ne)
    for i in range(ne):
        c = abs(ne/2.0-i)/(ne*.7)
        # let alle those turtles
        # make a step in parallel:
        for t in myturtles:
            t.rt(360./ne)
            t.pencolor(1-c,0,c)
            t.fd(sz)
```



4 Using turtles as geometrical objects

4.1 A few built-in shapes

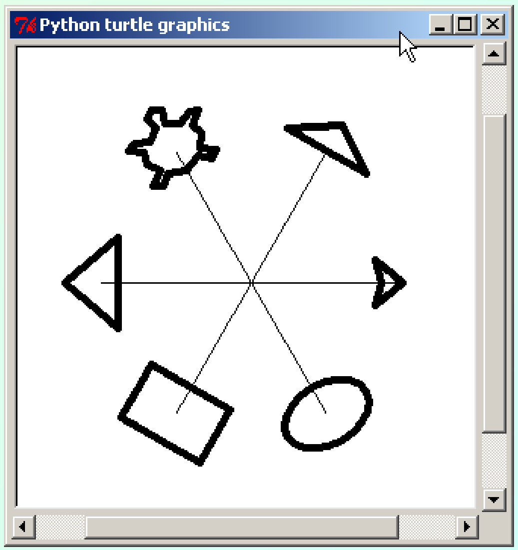
```
>>> from turtle import Screen, Turtle
>>> s = Screen()
>>> shapes = ["classic", "arrow", "turtle",
              "triangle", “square", "circle"]

>>> for i in range(6):,
           t = Turtle(shape=shapes[i])
           t.left(60*i)
           t.fd(100)
```



We can stretch/compress the turtles‘ shapes and control their outline widths using the method:

```
shapesize(stretch_wid, stretch_len, outline)
```

```
>>> for t in s.turtles():
          t.shapesize(3, 2, 5)
```

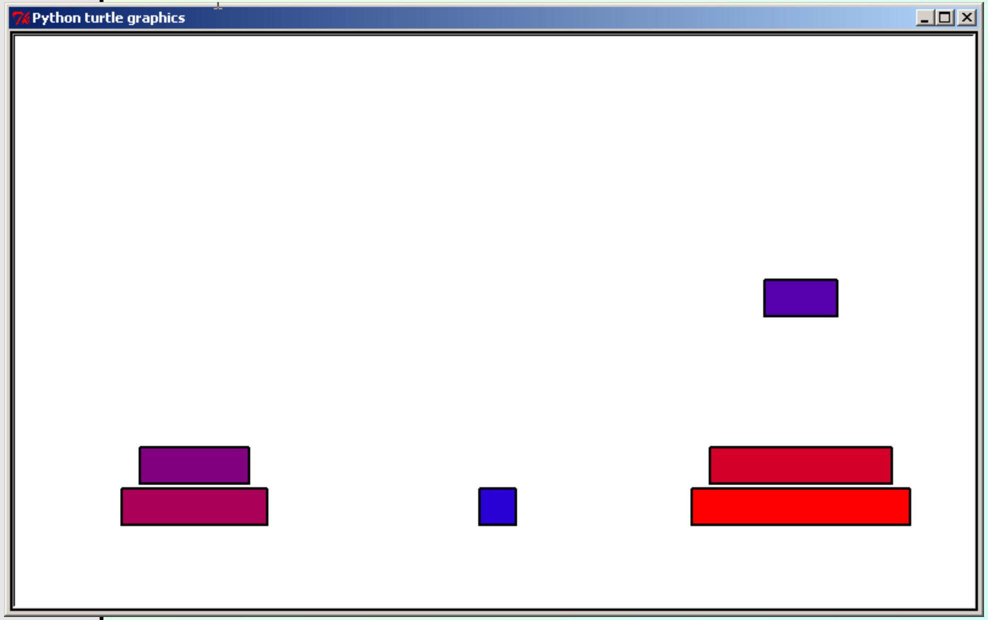


4 Using turtles as geometrical objects

4.2

4.2 Towers of Hanoi – Turtles as Hanoi-Discs

```
class Disc(Turtle):
    def __init__(self, n):
        Turtle.__init__(self, shape="square")
        self.pu()
        self.shapesize(1.5, n*1.5, 2)
        self.fillcolor(n/6., 0, 1-n/6.)

class Tower(list):
    def __init__(self, x):
        self.x = x
    def push(self, d):
        d.setx(self.x)
        d.sety(-150+34*len(self))
        self.append(d)
    def pop(self):
        d = list.pop(self)
        d.sety(150)
        return d

def hanoi(n, from_, with_, to_):
    if n > 0:
        hanoi(n-1, from_, to_, with_)
        to_.push(from_.pop())
        hanoi(n-1, with_, from_, to_)
```

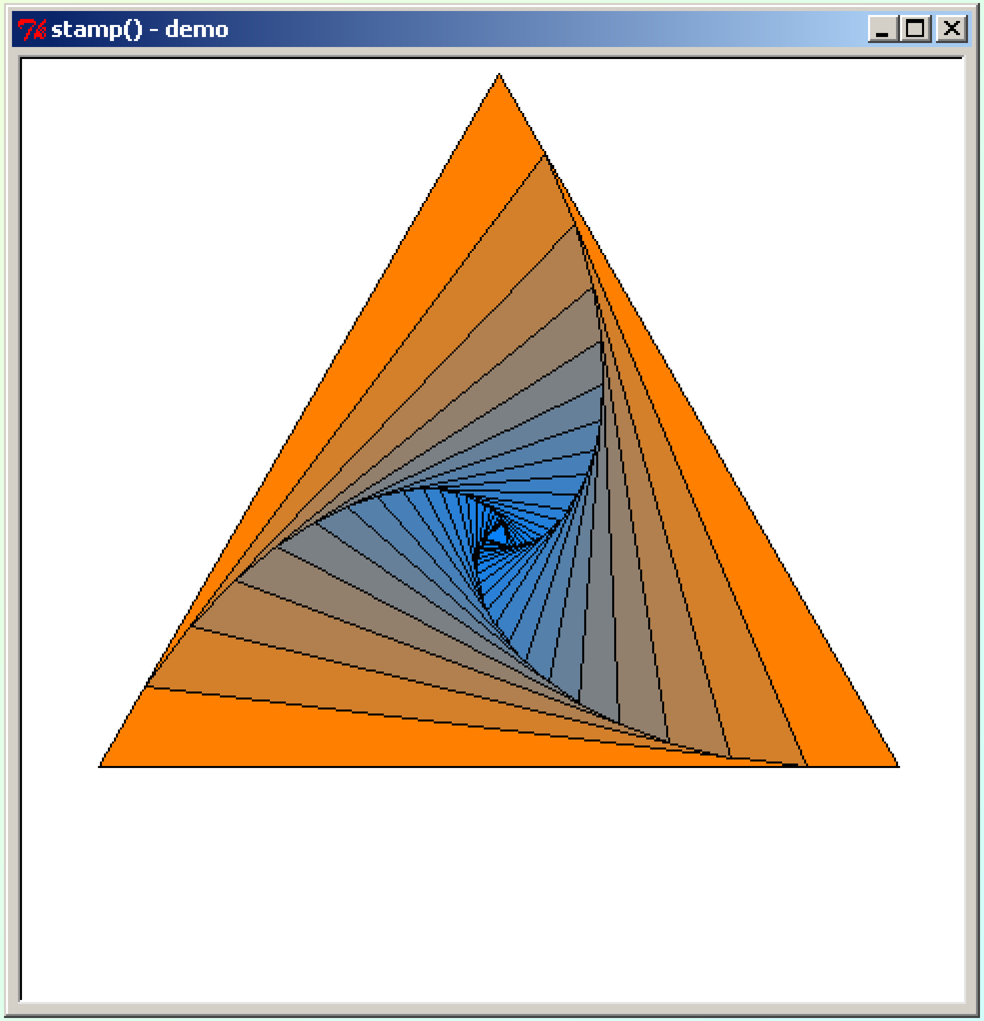


4 Using turtles as geometrical objects

4.3

4.3 `stamp()` - Example 1

```
>>> reset() #shape is "triangle"
>>> f = 0.83282
>>> phi = 6.89637
>>> s = 20
>>> c = 1
>>> for i in range(20):
        shapesize(s)
        fillcolor(c, 0.5, 1-c)
        dummy = stamp()
        s *= f
        c *= f
        right(phi)
```



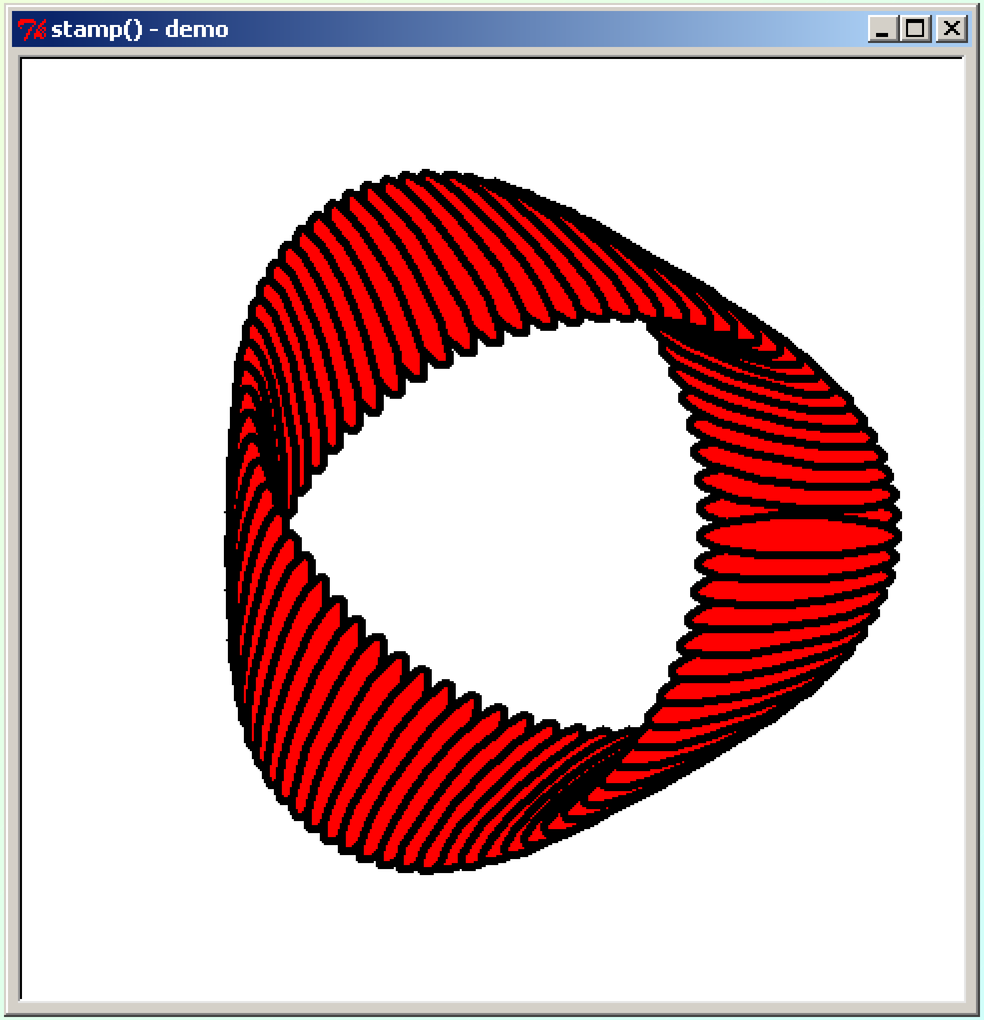
How would *you* compute appropriate values for **f** an **phi**?

4 Using turtles as geometrical objects

4.4

4.4 `stamp()` - Example 2: tilting shapes

```
>>> reset()
>>> shape("circle")
>>> pu(); goto(150,0)
>>> fillcolor("red")
>>> shapesize(5, 1, 4)
>>> for i in range(72):
     fd(12)
     lt(5)
     tilt(7.5)
     dummy = stamp()
```



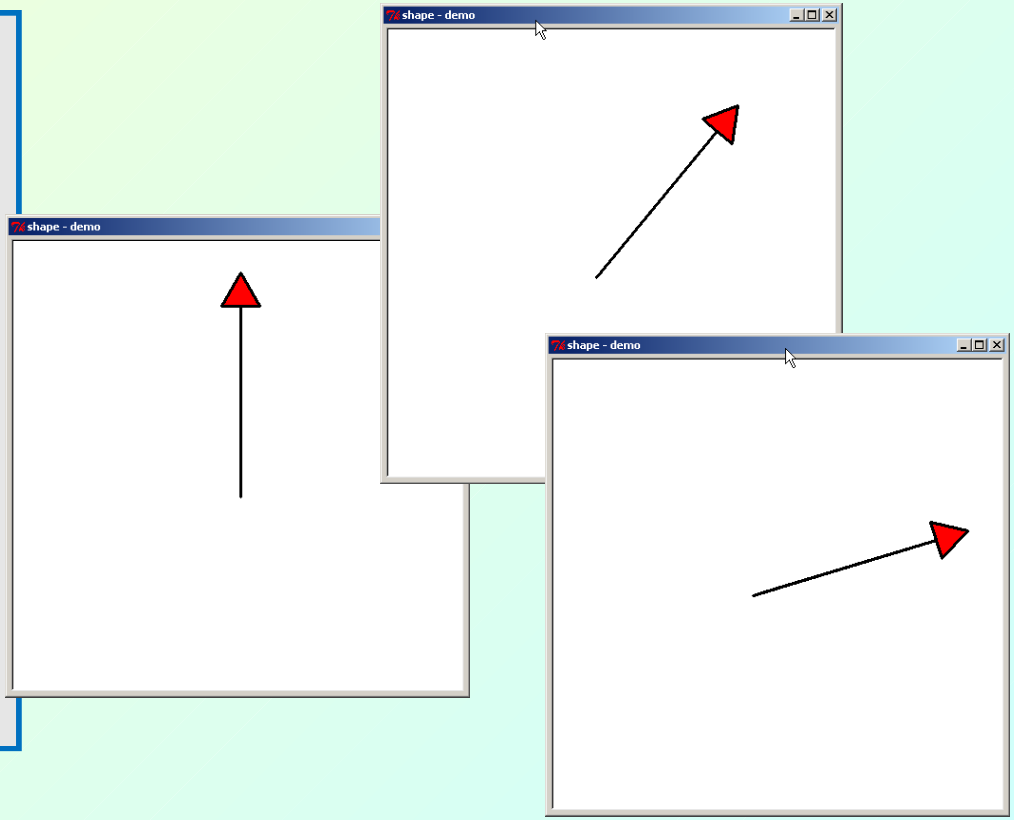
4 Using turtles as geometrical objects

4.5

4.5 Creating your own shapes

To create your own shape, you have somehow to generate a polygon to be used as the outline of the shape. (You may well use turtle graphics for this.) After having registered the new shape, you can use it:

```
>>> reset()
>>> begin_poly()
>>> bk(30)
>>> fd(200)
>>> lt(90); fd(a20); rt(120)
>>> fd(40); rt(120)
>>> fd(40); rt(120)
>>> fd(20)
>>> end_poly()
>>> p = get_poly()
>>> register_shape("clock_hand", p)
>>> reset()
>>> shape("clock_hand")
>>> fillcolor("red")
>>> shapesize(1, 1, 3)
>>> right(360)a
```



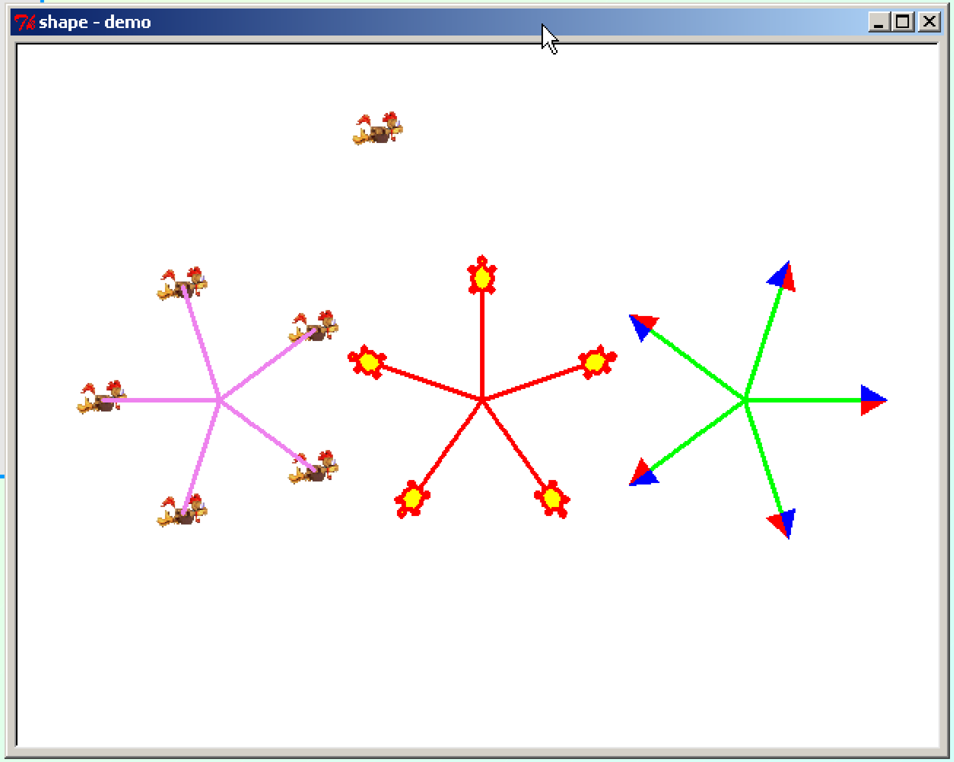
4 Using turtles as geometrical objects

4.6 compound shapes, gif-images as shapes

4.6

An example of creating a compound

```
begin_poly()
fd(10); lt(120); fd(20)
end_poly()
m1 = get_poly()
lt(120)
begin_poly()
fd(20); lt(120); fd(10)
end_poly()
m2=get_poly()
s = Shape("compound")  #use Shape class!
s.addcomponent(m1,"red")
s.addcomponent(m2,"blue")
register_shape("redblue", s)
```



Using a gif-image as shape:

```
register_shape("huhn01.gif")
```

Using these shapes:

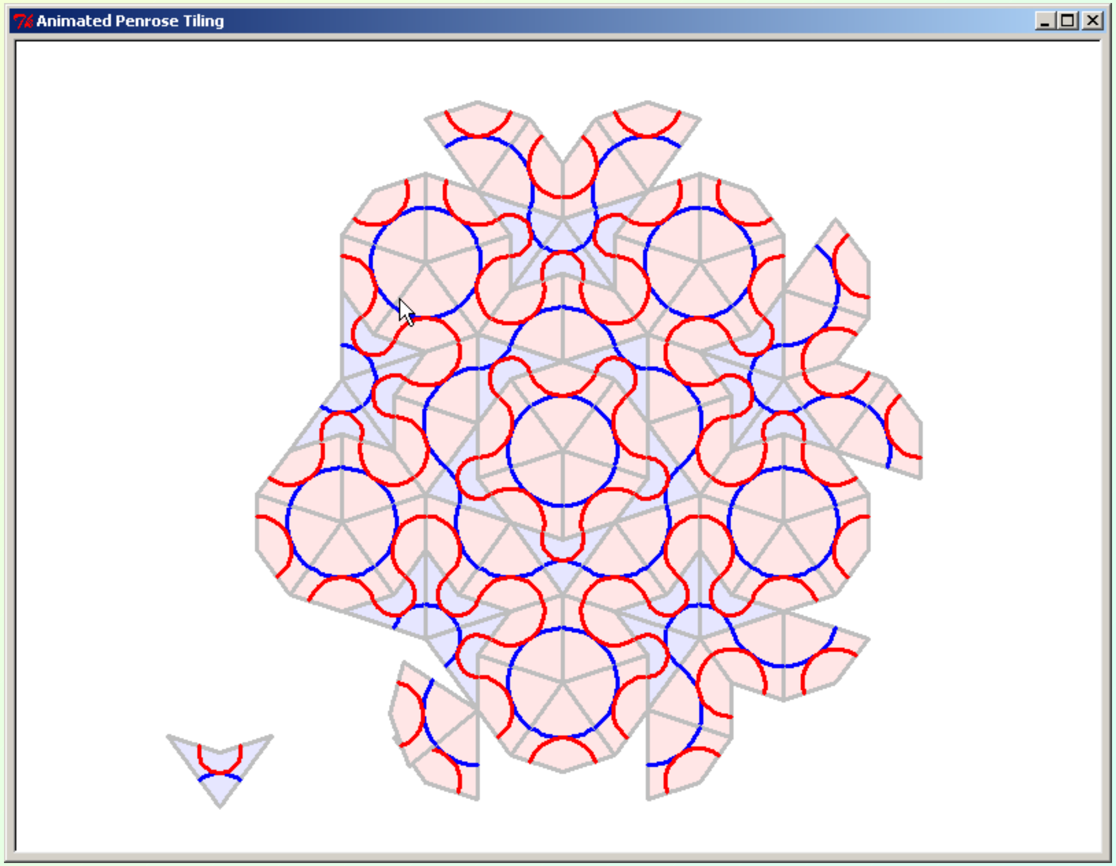
```
p = Turtle(shape="turtle")
q = Turtle(shape="redblue")
r = Turtle(shape="huhn01.gif")
```

4 Using turtles as geometrical objects

4.7

4.7 Penrose tiling

An example of an animated construction of a Penrose-tiling using compound shapes as Penrose-tiles (contributed by Wolfgang Urban):

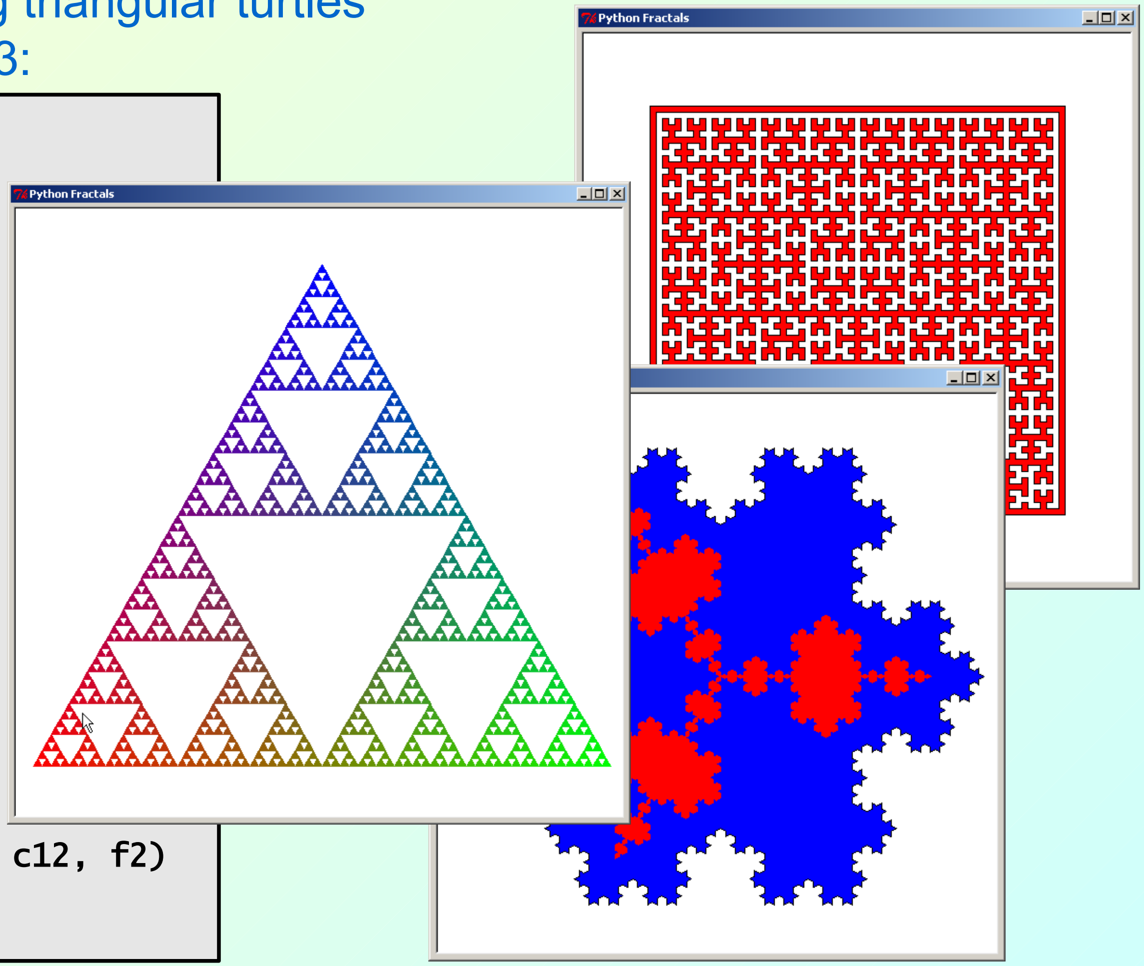


5 Classical turtle graphics playground (the Pythonian way)

5.1 Fractals, e. g. Hilbert, Koch, Sierpinsky

A Sierpinsky-script with stamping triangular turtles which uses color-vectors f1, f2, f3:

```
def SPT(laenge, depth, f1, f2, f3):
    if depth == 0:
        color((f1+f2+f3)/3)
        stamp() # triangular shape
    else:
        c12 = (f1+f2)/2
        c13 = (f1+f3)/2
        c23 = (f2+f3)/2
        fd(laenge)
        SPT(laenge/2, depth-1,
            c13, c23, f3)
        bk(laenge)
        left(120)
        fd(laenge)
        SPT(laenge/2, depth-1,
            c12, c13, f1)
        bk(laenge)
        lt(120)
        fd(laenge)
        SPT(laenge/2, depth-1, c23, c12, f2)
        bk(laenge)
        left(120)
```

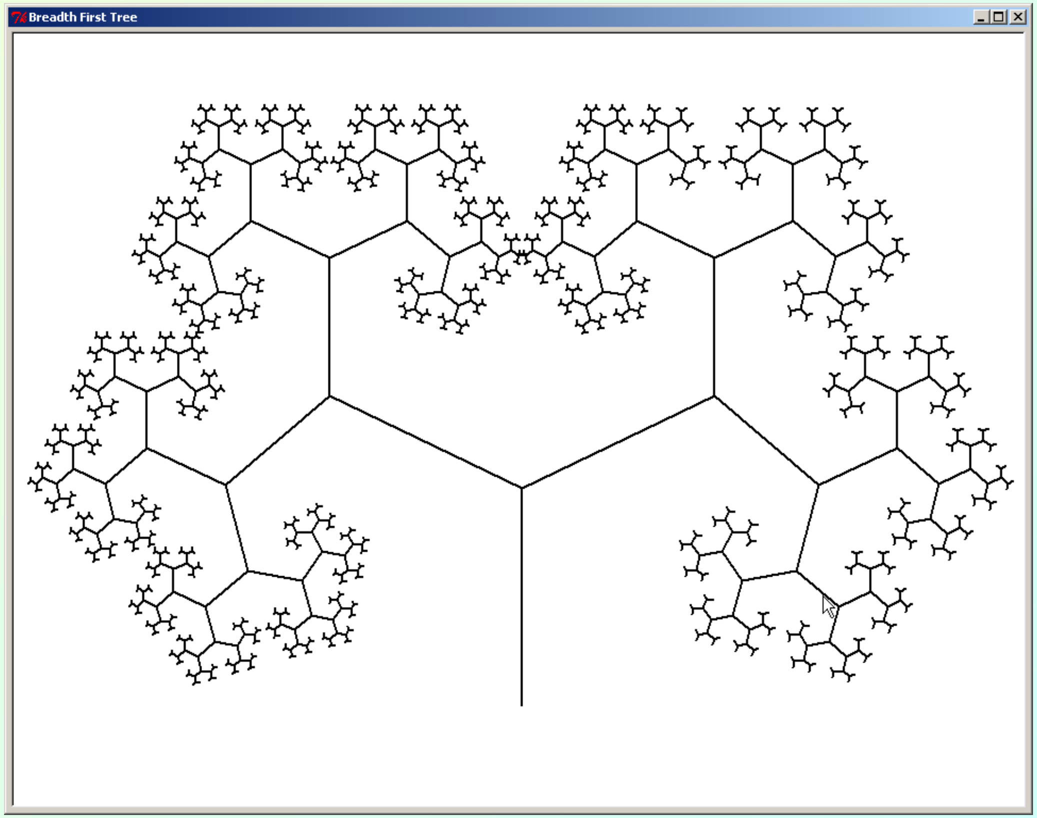


5 Classical turtle graphics playground

5.2

5.2 Breadth first tree (cloning turtles)

```
def tree(plist, l, a, f):
    while l > 3:
        qlist = []
        s.tracer(False)
        for p in plist:
            q = p.clone()
            p.left(a)
            p.forward(l)
            q.right(a)
            q.forward(l)
            qlist.append(5)
            qlist.append(q)
        s.tracer(True)
        Plist = qlist
        l = l*f

def maketree():
    p = Turtle(undobuffersize=0)
    p.ht(); p.speed(0)
    p.pensize(2)
    p.pu(); p.bk(260)
    p.pd(); p.fd(200)
    t = tree([p], 200, 65, 0.6375)
```



5 Classical turtle graphics playground

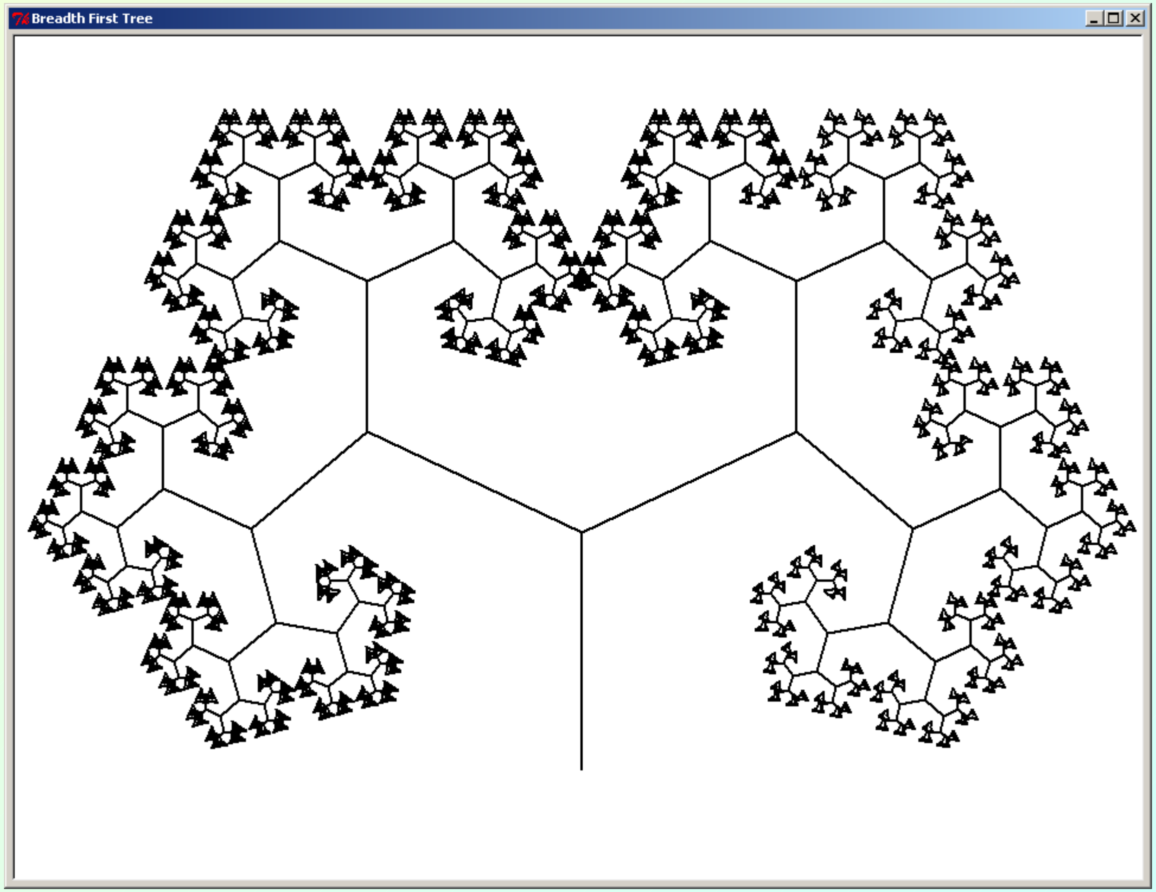
5.3 Breadth first tree (cloning turtles), (contd.)

We count the turtles:

```
>>> len(s.turtles())
1024
```

We look at them:

```
>>> for t in s.turtles():
        t.showturtle()

>>>
```



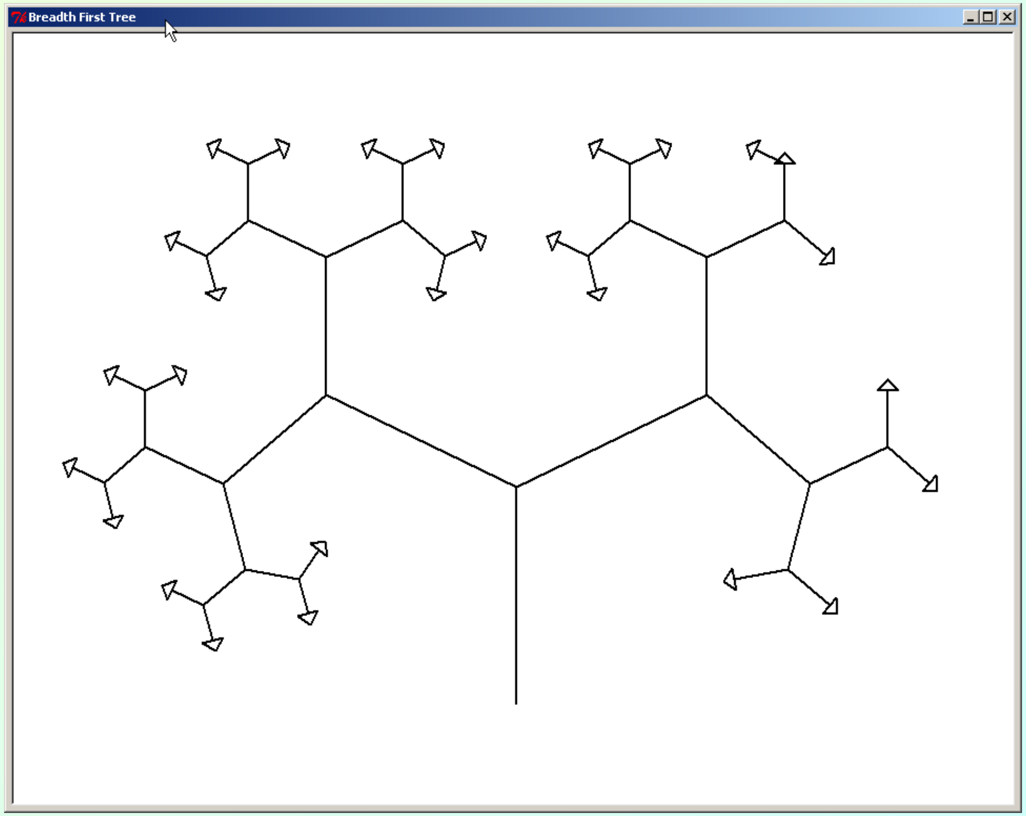
5 Classical turtle graphics playground

5.4

5.4 Breadth first tree (cloning turtles) (contd.)

How does that script work?
Let's have a look:

```
def tree(plist, l, a, f):
    while l > 30:
        qlist = []
        #s.tracer(False)
        for p in plist:
            q = p.clone()
            p.left(a)
            p.forward(l)
            q.right(a)
            q.forward(l)
            qlist.append(p)
            qlist.append(q)
        #s.tracer(True)
        plist = qlist
        l = l*f

def maketree():
    p = Turtle(undobuffersize=0)
    p.speed(1); p.pensize(2)
    #  and so on …
```



5 Classical turtle graphics playground

5.5 Lindenmayer Systems:

The biologist Aristid Lindenmayer devised a language to describe the structures of plants. An interpreter for this language is easily constructed in Python. Two of its main parts are:

A set of **standard symbols** and corresponding rules interpreting them:

```
        self.standardRules = {
             "-" : self.l,
             "+" : self.r,
             "|" : self.turn,
             "[" : self.save,
             "]" : self.load }

    # Standardrules
    def r(self):
        self.right(self.angle)

    def l(self):
        self.left(self.angle)

    # and more ...
```

An **interpreter** to execute these rules and also supplementary ones (using **`eval()`**). It is implemented as a generator:

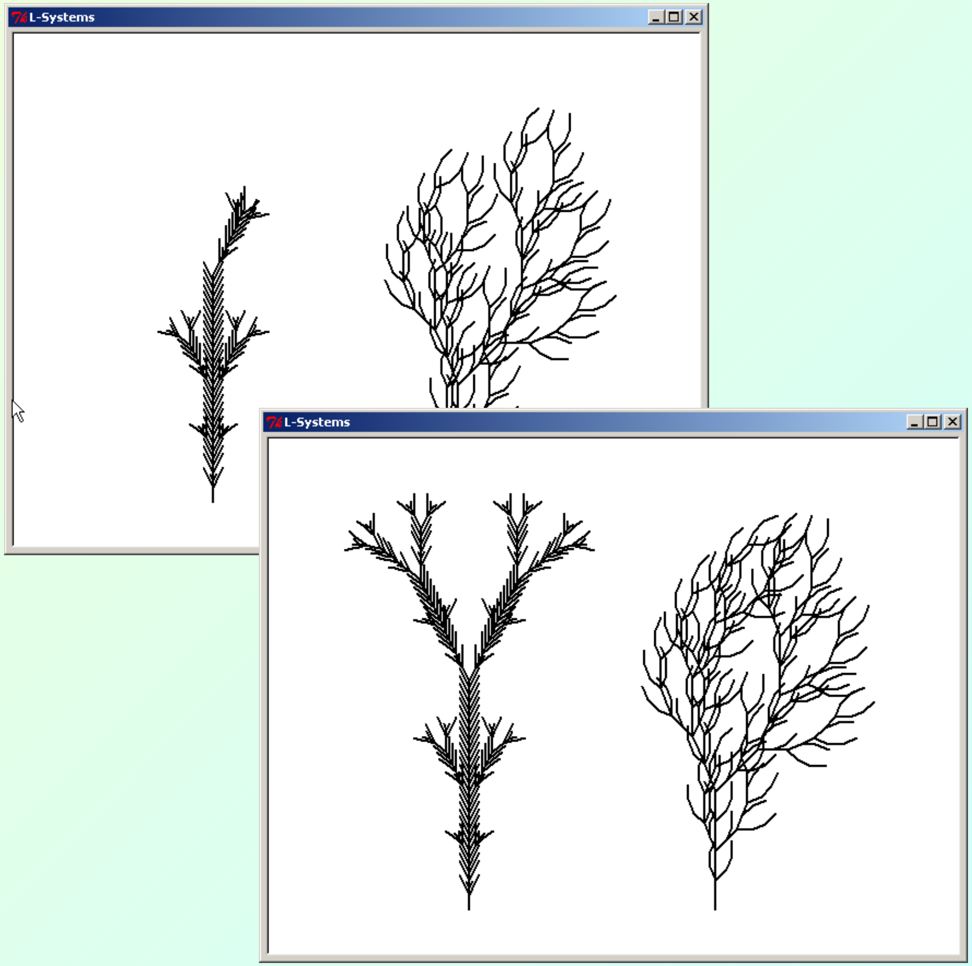
```
def _draw(self):
     i = 0
     for c in self.drawing:
         try:
             if hasattr(self,c):
                 eval("self.%s()" %c)
             else:
                 self.standardRules[c]()
         except KeyError:
             pass
         except:
             raise
         # We turn it into a generator!

         yield 1
```

5 Classical turtle graphics playground

5.6 Lindenmayer Systems (contd.): (1) Plants

Lindenmayer generators can make plants grow in parallel:

```
herb = { "axiom" : "G",
         "replacementRules" : {
            "G" : "GFX[+G][-G]",
            "X" : "X[-FFF][+FFF]FX"},
         "depth" : 5,
         "step" : 6.75,
         "angle" : 180.0/7,
         "startpos" : (-135, -192)}

# bush = { ... Another plant ...}

def main():
    l1 = LTurtle(**herb)
    l2 = LTurtle(**bush)
    l1.start(); l2.start()
    done = 0
    while done < 2:
        done = 0
        for l in l1, l2:
            try:
                next(l.draw)
            except StopIteration:
                done += 1
```



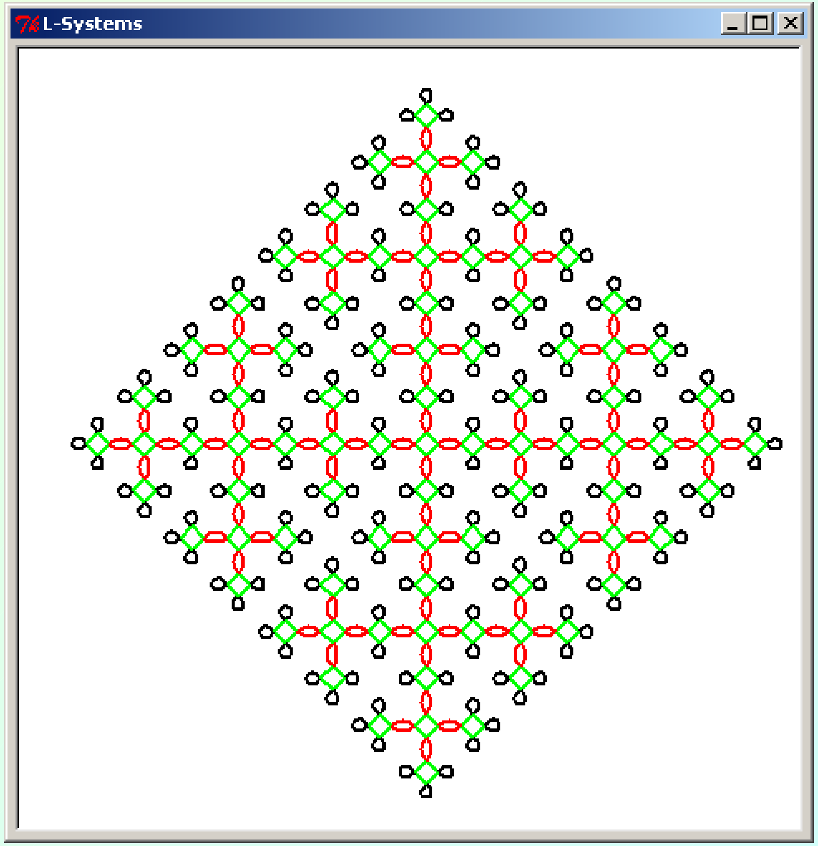
5 Classical turtle graphics playground

5.7

5.7 Lindenmayer Systems: (2) Ethno Mathematics:

Women in Tamil Nadu (India) create designs which can be described by Lindenmayer systems. Additional rules are needed:

```
anklets = {"axiom" : "GBGBGBGB",
  "replacementRules" : {"A" : "AGBGA",
                        "B" : "AGBGBGBGA" },
  "depth" : 3,
  "step" : 10,
  "angle" : 45,
  "startpos" : (0, 0),
  "startdir" : 45 }

# Additional Rules in Anklets, derived from
# LTurtle:
class Anklets(LTurtle):
    def A(self):
        self.pd()
        self.color("red")
        self.circle(self.step,90)
    def B(self):
        self.pd()
        self.color("black")
        # ... and so on
    def G(self):
        #... and so on
```

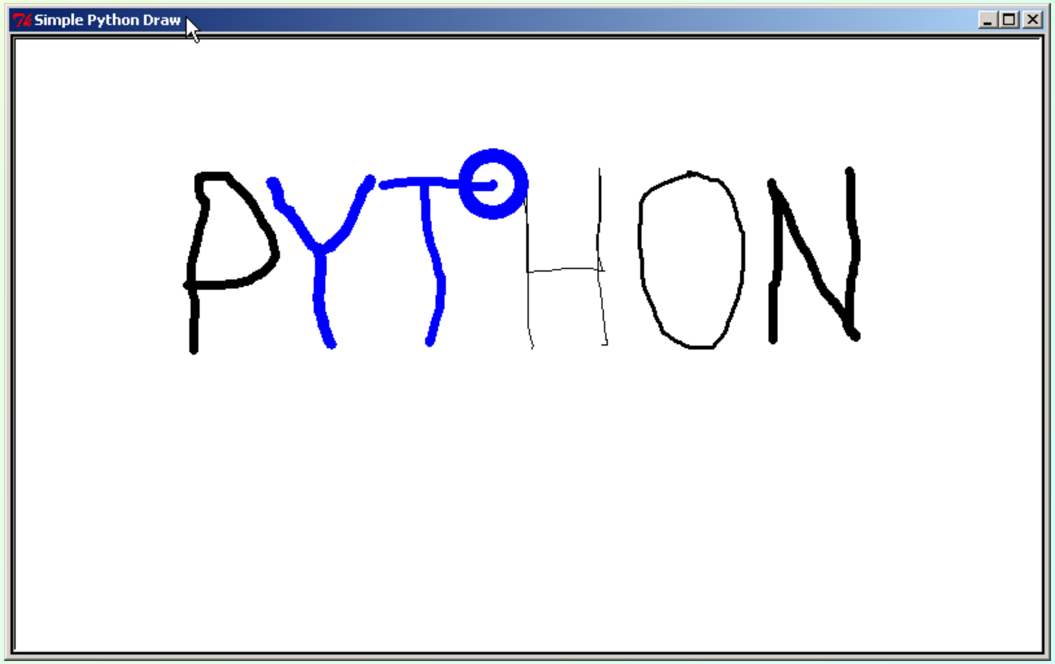


6 Creating Event Driven Programs

6.1

6.1 Some elements of a simple drawing program

```
>>> ondrag(goto)
>>> def jump(x, y):
    pu(); goto(x,y); pd()

>>> onscreenclick(jump)
>>> shape("circle")
>>> pensize(3)
>>> for n in "0123456789":
    onkey(lambda c=int(n):
            pensize(2*c+1), n)

>>> listen()
>>> speed(0)
>>> pencolor(„blue")
>>>
```



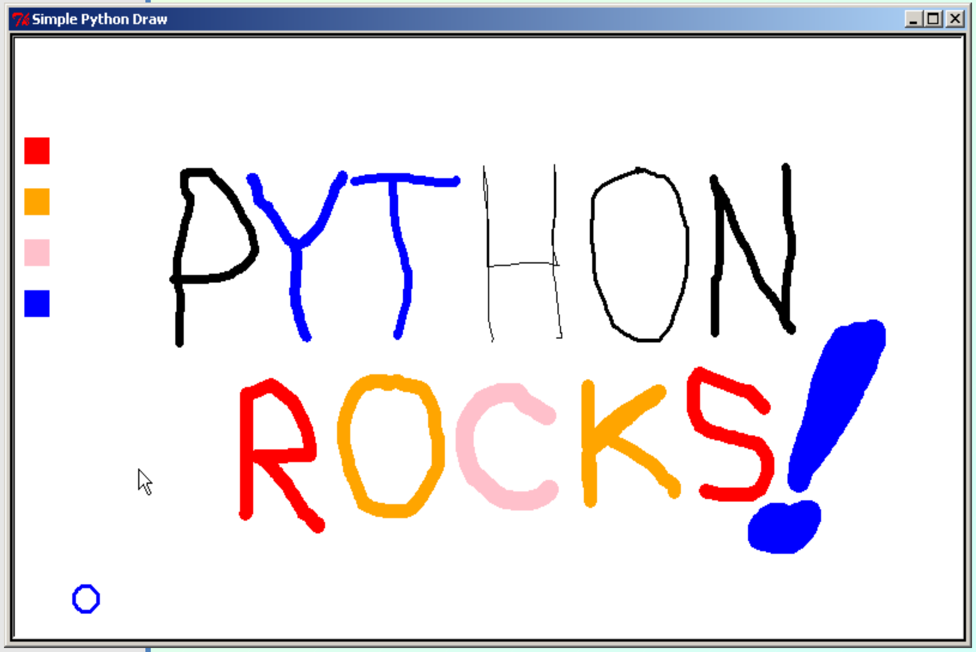
6 Creating Event Driven Programs

6.2 More elements of a simple drawing program

6.2

We define buttons to change the drawing turtle's pencolor

```
>>> class ColorButton(Turtle):
      def __init__(self, col, x, y):
        Turtle.__init__(self)
        self.pu(); self.goto(x, y)
        self.color(col)
        self.shape("square")
        self.onclick(self.setcolor)
      def setcolor(self, x, y):
        color(self.pencolor(),"")

>>> rb = ColorButton("red",
                     -380, 150)
>>> ob = ColorButton("orange",
                     -380, 110)
>>> def jump(x, y):
      if x > -350:
    pu(); goto(x,y); pd()

>>> onscreenclick(jump)
>>> pb = ColorButton("pink", -380, 70)
>>> bb = ColorButton("blue", -380, 30)
```



6 Creating Event Driven Programs

6.3 Animation via `ontimer`: a digital analog clock

6.3

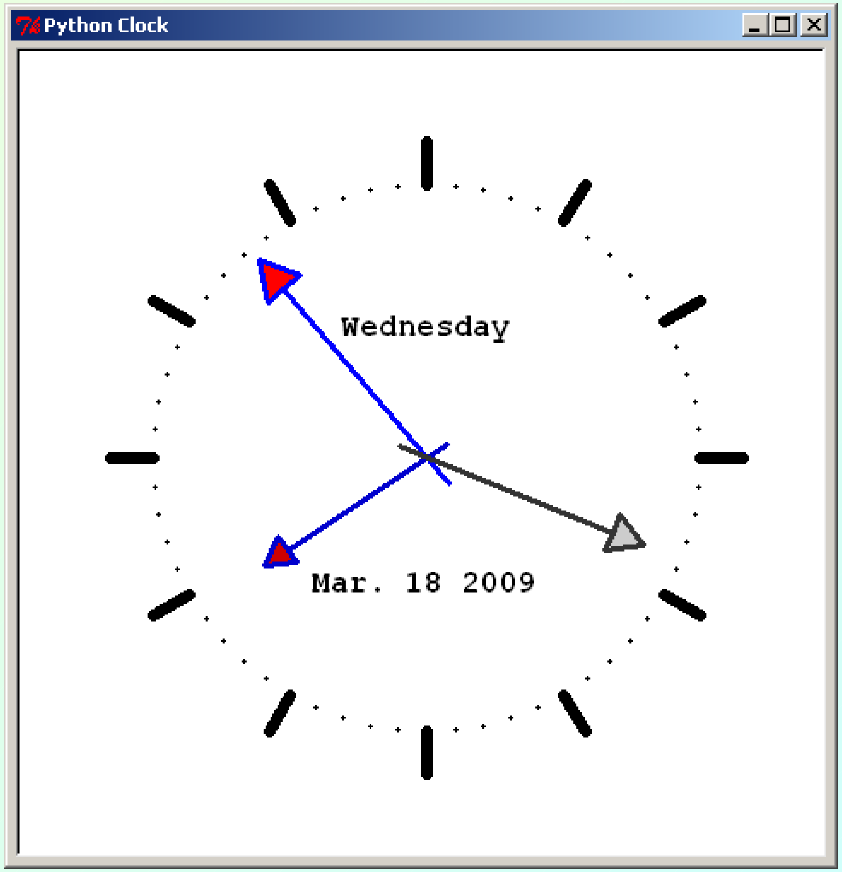
We use our clock hand (from) to construct a clock:

We use three clock hands:

```
second_hand = Turtle(shape="clock_hand")
second_hand.color("gray20", "gray80")
second_hand.shapesize(0.9, 0.9, 3)
minute_hand = Turtle(shape="clock_hand")
minute_hand.color("blue1", "red1")
minute_hand.shapesize(1, 1, 3)
hour_hand = Turtle(shape="clock_hand")
hour_hand.color("blue3", "red3")
hour_hand.shapesize(0.75, 0.75, 3)
```

2. We animate them using a timer:

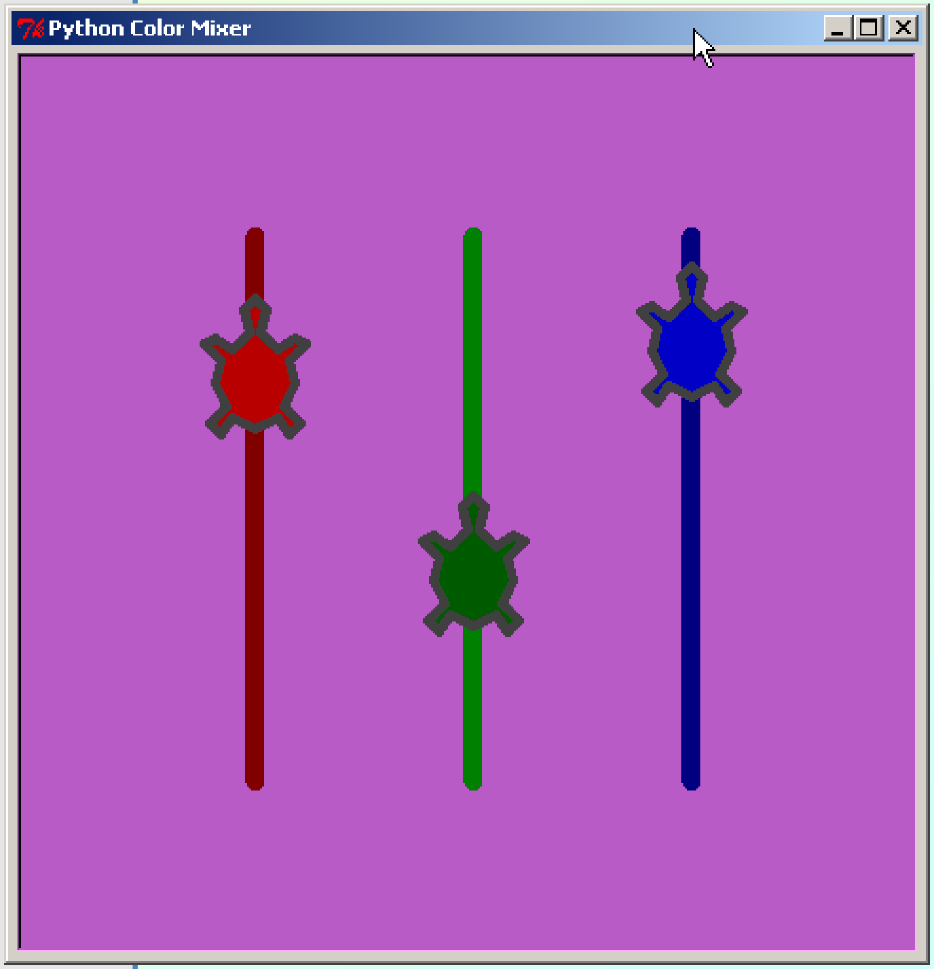
```
def tick():
    t = datetime.today()
    sekunde = t.second +t.microsecond*1e-6
    minute = t.minute + second/60.0
    hour = t.hour + minute/60.0
    # some stuff left out here
    second_hand.setheading(6*sekunde)
    minute_hand.setheading(6*minute)
    hour_hand.setheading(30*hour)
    ontimer(tick, 100)
```



6 Creating Event Driven Programs

6.4 **ondrag**: Color mixer (or, how to implement sliders)

```
class CTurtle(Turtle):

    def __init__(self, x):
        Turtle.__init__(self, shape="turtle")
        self.shapesize(3,3,5)
        self.speed(0)
        self.left(90)
        self.pensize(10)
        self._color = [0,0,0]
        self._color[x] = 0.5
        self.color(self._color)
        self.pu(); self.goto(x, 0)
        self.pd(); self.goto(x, 1)
        self.pu(); self.goto(x, .5)
        self.pencolor("gray25")
        self.ondrag(self.shift)

    def shift(self, x, y):
        self.sety(max(0,min(y,1)))
        x = self.xcor()
        self._color[x] = mix[x] = self.ycor()
        self.fillcolor(self._color)
        bgcolor(mix)

mix = [0.5, 0.5, 0.5]

setworldcoordinates(-1, -0.3, 3, 1.3)
delay(0)
red, green, blue = CTurtle(0), CTurtle(1), CTurtle(2)
bgcolor(mix)
```



7 Putting things together: creating games

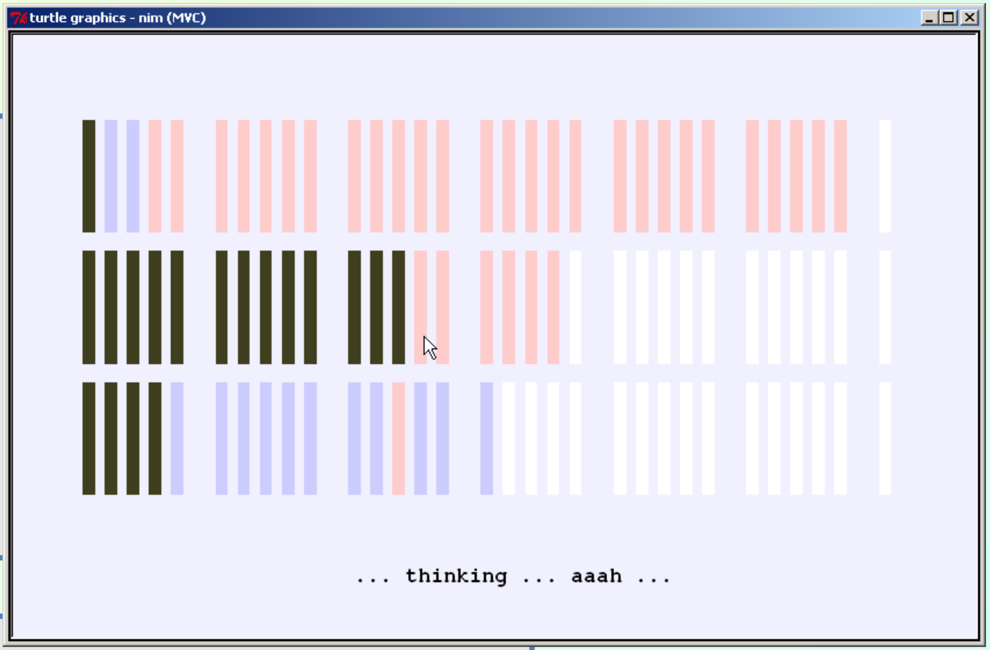
7.1 NIM

7.1

Turtles are used as nimsticks. They can easily display their state by changing colors. The player makes a move by clicking a stick.

The Stick class is part of the “View“:

```
class Stick(Turtle):
    def __init__(self, row, col):
        turtle.Turtle.__init__(self)
        self.row = row
        self.col = col
        self.shape("square")
        self.shapesize(STICKHEIGHT,
                       STICKWIDTH)
        self.color("white")
        self.pu()
        self.goto(6*(col//5) + col%5, -row)
```




```
class NimController(object):
    def __init__(self, game):
        self.game = game
        for stick in self.sticks.values():
            stick.onclick(stick.makemove)
        self.game.screen.onkey(self.game.model.setup, "space")
        self.game.screen.onkey(self.game.view.clear, "Escape")
        self.game.view.display("Press space bar to start game")
        self.game.screen.listen()
```

7 Putting things together: creating games

7.2 Tangram

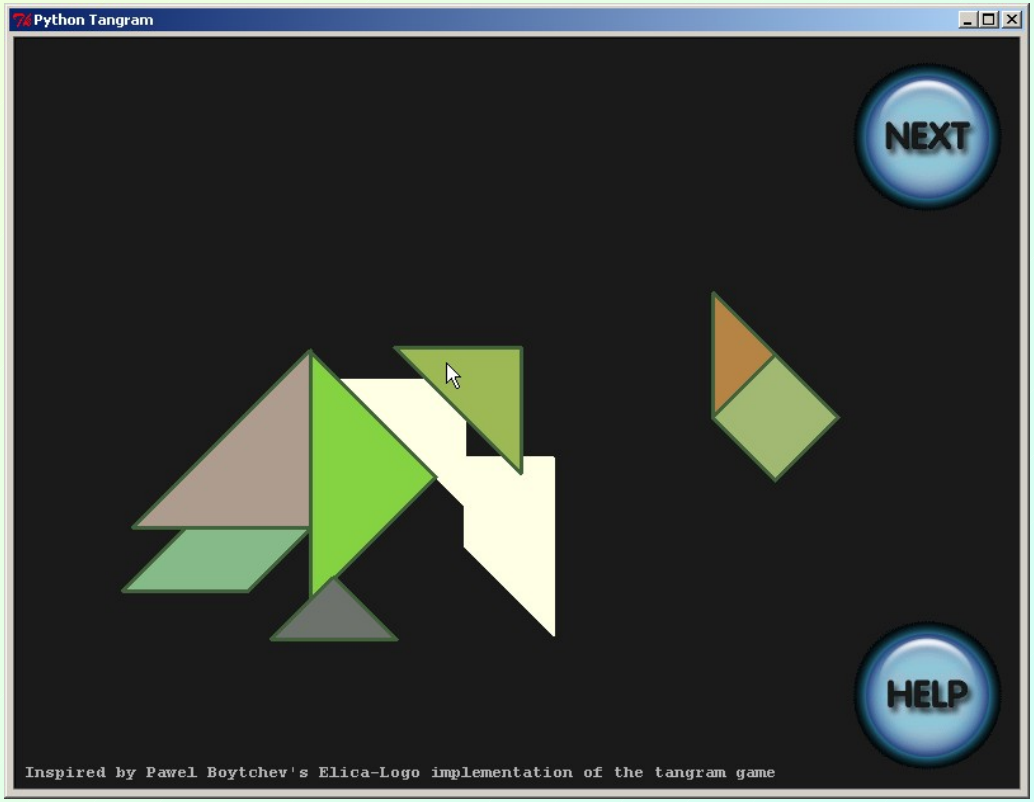
7.2

This example (approx. 230 lines) uses

(1) **Turtles as Tangram tiles**.
One special userdefined tile, the rhomboid, has two shapes: they can be changed by doubleclicking.

(2) **Turtles as Buttons**; their shapes are gif-images

It was inspired by an implementation by Pavel Boytchev, the creator of Elica-Logo. In a correspondence about this Python version he wrote:




```
Thanks, I checked the source of the rhomb. It looks really short. Actually, the whole tangram.py
is much shorter than I have expected ;)  -  Pavel
```

I think this is mostly due to the expressiveness of Python and also to some degree to the versatilty of the turtle module.

7 Putting things together: creating games

7.3

7.3 Moorhuhn

This game uses a gif as **background image**.

“Moorhuhn“s are clickable turtles with **gif images as shapes**, falling along parabolas when hit. (Again the Vec2D class for doing physics comes in handy here.)

(Re-) starting the game is controlled by a key-event.

A bit of **native Tkinter-stuff** (for the cross cursor) and sound is added.

8 Where to go from here?

8.1

8.1 Feature requests?

General remark: the API of the turtle module is already rather large. So features should only be added after careful consideration.

Up to now, users have requested the following:

- Some sort of **input dialog**. - In the current version, input can only be done within a terminal (or IDLE) or by using native Tkinter dialogs.
- Using **more types of events** (e. g. keyPressed additionally to keyRelease)
- **Enhancing the demoViewer** with capabilities to trace the execution of turtle graphics scripts
- Creating a user interface which combines an IDLE shell window with a turtle graphics window

8 Where to go from here?

8.2

8.2 Open issue (from my point of view): How to implement the singleton Screen()

During the last weeks before adopting turtle.py into the standard library, there came up a discussion on this point, resulting in the

- Current implementation: Screen() is implemented as a function which returns the **single** instance of a _Screen() object that can be constructed.

- Alternative Approach: Implementing Screen() using some sort of singleton design pattern (e. g. the Borg idiom). **Advantage: Screen() could be subclassed to create, for instance, a GameScreen() class etc.**

N B: Whether subclassing singletons makes sense or not is vehemently being discussed among computer scientists. This discussion has revealed very controversial opinions.

Which alternative is preferable should be assessed along didactical criteria.

8 Where to go from here?

8.3 Implementing turtle.py for different graphics toolkits/libraries

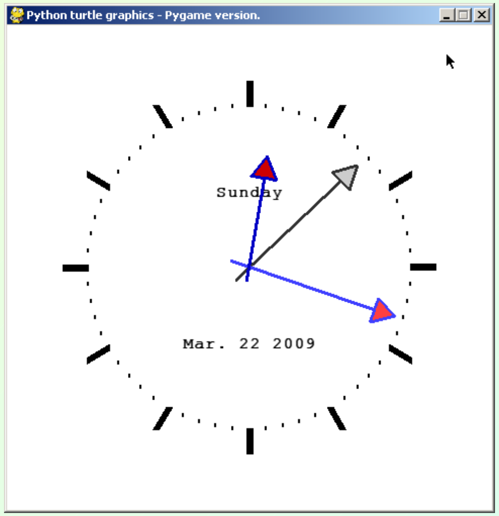
8.3

The turtle module is designed in a way so that essentially all of the turtle graphics machinery is based on a class `TurtleScreenBase`, which provides the interface to the underlying graphics toolkit Tkinter.

So it's easy to port turtle.py to different graphics toolkits/libraries, simply by replacing this Tkinter base class with an appropriate different one.

I've done two ports:

Pygame:



Jython: