Recursive Images

1. A Simple GUI
   - a regular n-gon
   - recursive images

2. The Cantor Set
   - recursive definition
   - recursive drawing algorithm

3. The Koch Curve and Flake
   - recursive drawing algorithm
   - GUI for a Koch flake

4. Space Filling Curves and L-systems
   - a GUI for Hilbert curves
   - implementing the moves
Recursive Images

1. A Simple GUI
   - a regular n-gon
   - recursive images

2. The Cantor Set
   - recursive definition
   - recursive drawing algorithm

3. The Koch Curve and Flake
   - recursive drawing algorithm
   - GUI for a Koch flake

4. Space Filling Curves and L-systems
   - a GUI for Hilbert curves
   - implementing the moves
A Regular n-Gon
the class **MultiGon** – write doc strings first

class MultiGon(object):
    
    GUI to draw a regular n-gon on canvas.
    
def __init__(self, wdw):
        
        Determines the layout of the GUI.
        
def draw_gon(self, val):
            
            Draws a regular n-gon.
            
def main():
            
            Instantiates the GUI object
            and launches the main event loop.
the layout of the GUI

def __init__(self, wdw):
    """
    Determines the layout of the GUI.
    """
    wdw.title('regular n-gon')
    self.dim = 400
    self.ngn = IntVar()
    self.scl = Scale(wdw, orient='horizontal',
                    from_=1, to=20, tickinterval=1,
                    length=self.dim, variable=self.ngn,
                    command=self.draw_gon)
    self.scl.set(10)
    self.scl.grid(row=0, column=0)
    self.cnv = Canvas(wdw, width=self.dim,
                      height=self.dim, bg='white')
    self.cnv.grid(row=1, column=0)
the function `draw_gon()`

the Scale triggers the command `draw_gon()` and determines the data attribute `ngn` of a MultiGon object.

The second argument `val` is the value for `ngn` passed to `draw_gon()` via the Scale.

def draw_gon(self, val):
    """
    Draws a regular n-gon.
    """
    xctr = self.dim/2
    yctr = self.dim/2
    radius = 0.4*self.dim
    self.cnv.delete(ALL)
    self.cnv.create_text(xctr, yctr, text=val, tags="text")
    ngn = int(val)

The value `val` is passed as a string.
from math import cos, sin, pi

All $n$ points lie on a circle, equispaced using angle $2\pi/n$.

```python
pts = []
for i in range(0, ngn):
    xpt = xctr + radius*cos(2*i*pi/ngn)
    ypt = yctr + radius*sin(2*i*pi/ngn)
    self.cnv.create_oval(xpt-6, ypt-6, xpt+6, ypt+6, 
        width=1, outline='black', fill='SkyBlue2', 
        tags="dot")
pts.append((xpt, ypt))
for i in range(0, ngn-1):
    self.cnv.create_line(pts[i][0], pts[i][1], 
        pts[i+1][0], pts[i+1][1], width=2)
self.cnv.create_line(pts[ngn-1][0], pts[ngn-1][1], 
    pts[0][0], pts[0][1], width=2)
```
Recursive Images

1. A Simple GUI
   - a regular n-gon
   - recursive images

2. The Cantor Set
   - recursive definition
   - recursive drawing algorithm

3. The Koch Curve and Flake
   - recursive drawing algorithm
   - GUI for a Koch flake

4. Space Filling Curves and L-systems
   - a GUI for Hilbert curves
   - implementing the moves
a Cantor set
a Koch curve
a Koch flake
a Sierpinski gasket
a GUI for Hilbert curves
Recursive Images

1. A Simple GUI
   - a regular n-gon
   - recursive images

2. The Cantor Set
   - recursive definition
   - recursive drawing algorithm

3. The Koch Curve and Flake
   - recursive drawing algorithm
   - GUI for a Koch flake

4. Space Filling Curves and L-systems
   - a GUI for Hilbert curves
   - implementing the moves
The Cantor set is defined by three rules

1. take the interval \([0, 1]\),
2. remove the middle part third of the interval,
3. repeat rule 2 on the first and third part.

The Cantor set is infinite, to visualize at level \(n\):

- \(n = 0\): start at \([0, 1]\),
- \(n > 0\): apply rule 2 \(n\) times.
GUI for a Cantor Set

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- clear canvas
the class CantorSet – write doc strings first

class CantorSet(object):
    """
    GUI to draw a Cantor set on canvas.
    """
    def __init__(self, wdw, N):
        """
        A Cantor set with N levels.
        """
    def cantor(self, lft, rgt, hgt, txt, lvl):
        """
        Draws a line from lft to rgt, at height hgt, txt is a string, int(txt) equals the number of times the middle third must be removed, lvl is the level of recursion.
        """
    def draw_set(self, val):
        """
        Draws a Cantor set.
        """
the layout of the GUI

def __init__(self, wdw, N):
    
    """
    A Cantor set with N levels.
    """

    wdw.title('a cantor set')
    self.dim = 3**N+20
    self.nsc = IntVar()
    self.scl = Scale(wdw, orient='horizontal',
                     from_=0, to=N, tickinterval=1, length=self.dim,
                     variable=self.nsc, command=self.draw_set)
    self.scl.grid(row=0, column=0)
    self.scl.set(0)
    self.cnv = Canvas(wdw, width=self.dim, 
                     height=self.dim/3, bg='white')
    self.cnv.grid(row=1, column=0)
    self.btt = Button(wdw, text="clear canvas", 
                     command=self.clear_canvas)
    self.btt.grid(row=2, column=0)
The method `clear_canvas()` is triggered by the button.

```python
def clear_canvas(self):
    """
    Clears the entire canvas.
    """
    self.cnv.delete(ALL)
```
The method `draw_set()` is triggered by the scale.

```python
def draw_set(self, val):
    """
    Draws a Cantor set.
    """
    nbr = int(val)
    self.cantor(10, self.dim-10, 30, val, nbr)
```

The method `cantor()` is recursive.
Recursive Images

1. A Simple GUI
   - a regular n-gon
   - recursive images

2. The Cantor Set
   - recursive definition
   - recursive drawing algorithm

3. The Koch Curve and Flake
   - recursive drawing algorithm
   - GUI for a Koch flake

4. Space Filling Curves and L-systems
   - a GUI for Hilbert curves
   - implementing the moves
def cantor(self, lft, rgt, hgt, txt, lvl):
    """
    Draws a line from lft to rgt, at height hgt; txt is a string, int(txt) equals the number of times the middle third must be removed; lvl is the level of recursion.
    """

The parameters lft, rgt, and hgt define the line segment from (lft, hgt) to (rgt, hgt).

The parameter txt is the value passed via the Scale, as text string, txt is also put on Canvas.

Initially: lvl = int(txt).
With every recursive call, lvl is decremented by 1.
The `lvl` in `cantor(self, lft, rgt, hgt, txt, lvl)` controls the recursion.

At `lvl = 0`, the line segment from `(lft, hgt)` to `(rgt, hgt)` is drawn.

For `lvl > 0`, we compute left and right limit of the middle third of `[lft, rgt]`, respectively denoted by `nlf` and `nrg` as

\[
\begin{align*}
nlf &= lft + \frac{rgt - lft}{3} = \frac{2lft + rgt}{3} \\
nrg &= rgt = \frac{rgt - lft}{3} = \frac{lft + 2rgt}{3}
\end{align*}
\]

Then we make two recursive calls:

\[
\begin{align*}
\text{self.cantor}(lft, nlf, hgt+30, txt, lvl-1) \\
\text{self.cantor}(nrg, rgt, hgt+30, txt, lvl-1)
\end{align*}
\]
def cantor(self, lft, rgt, hgt, txt, lvl):

    if lvl == 0:  # draw line segment
        self.cnv.create_line(lft, hgt, rgt, hgt, width=2)
    else:
        nlf = (2*lft+rgt)/3
        nrg = (lft+2*rgt)/3
        self.cantor(lft, nlf, hgt+30, txt, lvl-1)
        self.cantor(nrg, rgt, hgt+30, txt, lvl-1)
    if lvl == int(txt):  # put text string
        xctr = self.dim/2
        if txt == '0':
            self.cnv.create_text(xctr, hgt-10, text=txt)
        else:
            self.cnv.create_text(xctr, hgt+lvl*30, \ 
                                  text=txt)
The Koch curve is a Cantor set where the removed middle third of the interval is replaced by a wedge.

The top of the wedge is above the midpoint of the removed middle interval.

The slopes of the wedge make an angle of 60 degrees with respect to the rest of the line segment.

To visualize a Koch curve at level $n$:

- $n = 0$: start at $[0, 1]$,
- $n > 0$: make wedges $n$ times.
GUI for a Koch Curve
class KochCurve(object):
    
    GUI to draw a Koch curve on canvas.
    
def __init__(self, wdw, N):
        
        A Koch curve with N levels.
    
def koch(self, left, right, k):
        
        A Koch curve from left to right with k levels.
    
def draw_curve(self, val):
        
        Draws a Koch curve.
the layout of the GUI

def __init__(self, wdw, N):
    """
    A Koch curve with N levels.
    """

    wdw.title('a Koch curve')
    self.dim = 3**N+20
    self.nvr = IntVar()
    self.scl = Scale(wdw, orient='horizontal',
        from_=0, to=N, tickinterval=1,
        length=self.dim, variable=self.nvr,
        command=self.draw_curve)
    self.scl.set(0)
    self.scl.grid(row=0, column=0)
    self.cnv = Canvas(wdw, width=self.dim,
        height=self.dim/3, bg='white')
    self.cnv.grid(row=1, column=0)
    self.btt = Button(wdw, text="clear canvas",
        command=self.clear_canvas)
    self.btt.grid(row=2, column=0)
The method `clear_canvas()` is triggered by a button.

```python
def clear_canvas(self):
    ""
    Clears the entire canvas.
    ""
    self.cnv.delete(ALL)
```
The method \texttt{draw\_curve()} is activated by a scale.

```python
def draw\_curve(self, val):
    '''
    Draws a Koch curve.
    '''
    nbr = int(val)
    left = (10, self.dim/3-20)
    right = (self.dim-10, self.dim/3-20)
    self.koch(left, right, nbr)
```

The method \texttt{koch()} is recursive, to draw a Koch curve of \texttt{nbr} levels from \texttt{left} to \texttt{right}.  

The method \texttt{draw\_curve()} is activated by a scale.
Recursive Images

1. A Simple GUI
   - a regular n-gon
   - recursive images

2. The Cantor Set
   - recursive definition
   - recursive drawing algorithm

3. The Koch Curve and Flake
   - recursive drawing algorithm
   - GUI for a Koch flake

4. Space Filling Curves and L-systems
   - a GUI for Hilbert curves
   - implementing the moves
recursive drawing algorithm

The `nbr in koch(left, right, nbr)` controls the recursion.

For `nbr = 0`, the line segment from `left` to `right` is drawn.

For `nbr > 0`, we compute `left nlf`, `right nrg`, and midpoint `mid` of the middle third of line segment.

Angle of 60 degrees: $\sin\left(\frac{\pi}{3}\right) = \frac{\sqrt{3}}{2}$ is height, multiplied with $\frac{1}{3}$th of length of segment.

The peak of the wedge is relative to the size of the interval and the position of the midpoint `mid`:

$$
top = \left(\frac{\sqrt{3}}{6}(left[1] - right[1]), \frac{\sqrt{3}}{6}(right[0] - left[0])\right),$$

$$peak = (mid[0] - top[0], mid[1] - top[1])$$

Recall that on the canvas: (0,0) is at the topmost left corner.
code for the method \texttt{koch()}

def koch(self, left, right, k):
   """
   A Koch curve from left to right with k levels.
   """
   if k == 0:
       self.cnv.create_line(left[0], left[1],
                           right[0], right[1], width=2)
   else:
       nlf = ((2*left[0]+right[0])/3.0, (2*left[1]+right[1])/3.0)
       nrg = ((left[0]+2*right[0])/3.0, (left[1]+2*right[1])/3.0)
       mid = ((left[0]+right[0])/2.0, (left[1]+right[1])/2.0)
       ratio = sqrt(3)/6
       top = (ratio*(left[1]-right[1]), ratio*(right[0]-left[0]))
       peak = (mid[0]-top[0], mid[1]-top[1])
       self.koch(left, nlf, k-1)
       self.koch(nlf, peak, k-1)
       self.koch(peak, nrg, k-1)
       self.koch(nrg, right, k-1)
Recursive Images

1. A Simple GUI
   - a regular n-gon
   - recursive images

2. The Cantor Set
   - recursive definition
   - recursive drawing algorithm

3. The Koch Curve and Flake
   - recursive drawing algorithm
   - GUI for a Koch flake

4. Space Filling Curves and L-systems
   - a GUI for Hilbert curves
   - implementing the moves
GUI for a Koch flake

![Koch flake GUI](image)
the class **KochFlake** – write doc strings first

class KochFlake(object):
    
    """
    GUI to draw a Koch flake canvas.
    """

def __init__(self, wdw):
    
    """
    Determines the layout of the GUI.
    """

def koch(self, left, right, k):
    
    """
    A Koch flake from left to right with k levels.
    """

def draw_flake(self, val):
    
    """
    Draws a regular Koch flake.
    """
the scales in \texttt{\_\_init\_}

def \texttt{\_\_init\_}(self, wdw):

self.nbr = IntVar()
self.k = IntVar()
self.scn = Scale(wdw, orient='horizontal', 
                 from_=3, to=20, tickinterval=1, 
                 length=self.dim, variable=self.nbr, 
                 command=self.draw_flake)
self.scn.set(10)
self.scn.grid(row=0, column=1)
self.sck = Scale(wdw, orient='vertical', 
                 from_=0, to=6, tickinterval=1,
                 length=self.dim, variable=self.k,
                 command=self.draw_flake)
self.sck.set(0)
self.sck.grid(row=1, column=0)
def draw_flake(self, val):
    """
    Draws a regular Koch flake.
    """
    (xctr, yctr) = (self.dim/2, self.dim/2)
    radius = 0.4*self.dim
    self.cnv.delete(ALL)
    nbr = self.nbr.get()
    k = self.k.get()
    txt = '(' + str(nbr) + ',' + str(k) + ')
    self.cnv.create_text(xctr, yctr, text=txt, tags="text")
    pts = []
    for i in range(0, nbr):
        xpt = xctr + radius*cos(2*i*pi/nbr)
        ypt = yctr + radius*sin(2*i*pi/nbr)
        pts.append((xpt, ypt))
    for i in range(0, nbr-1):
        self.koch(pts[i], pts[i+1], k)
    self.koch(pts[nbr-1], pts[0], k)
Space Filling Curves

production rules for Hilbert curves

$H_1$

$H_2$

$H_3$

moves

recursion

A : D ← A ↓ A → B
B : C ↑ B → B ↓ A
C : B → C ↑ C ← B
D : A ↓ D ← D ↑ C
Recursive Images

1. A Simple GUI
   - a regular n-gon
   - recursive images

2. The Cantor Set
   - recursive definition
   - recursive drawing algorithm

3. The Koch Curve and Flake
   - recursive drawing algorithm
   - GUI for a Koch flake

4. Space Filling Curves and L-systems
   - a GUI for Hilbert curves
   - implementing the moves
A GUI for Hilbert Curves
A GUI for Hilbert Curves – write doc strings first

class Hilbert:
    
    """
    GUI for Hilbert’s space filling curves.
    """
    def __init__(self, wdw, dimension):
        """
        Defines a canvas, buttons, and scale.
        """
    def plot(self, nxp, nyp):
        """
        Plots a line for the current position
to the new coordinates (nxp, nyp), and
sets the current position to (nxp, nyp).
        """
    def draw(self, val):
        """
        Draws the Hilbert curve on canvas.
        """
The Moves in the GUI

def amove(self, k):
    """
    Turns counterclockwise from up right to down right.
    """

def bmove(self, k):
    """
    Turns clockwise from down left to down right.
    """

def cmove(self, k):
    """
    Turns counterclockwise from down left to up left.
    """

def dmove(self, k):
    """
    Turns clockwise from up right to up left.
    """
the method `plot()`

```python
def plot(self, nxp, nyp):
    """
    Plots a line for the current position to the new coordinates (nxp, nyp), and sets the current position to (nxp, nyp).
    """
    self.cnv.create_line(self.xpt, self.ypt, nxp, nyp, width=2)
    self.xpt = nxp
    self.ypt = nyp
```
the method draw

def draw(self, val):
    """
    Draws the Hilbert curve on canvas.
    """
    nbr = int(val)
    self.hgt = self.dim/2**nbr
    self.ypt = 5 + self.dim - self.hgt/2
    self.xpt = self.ypt
    self.amove(nbr)
Recursive Images

1. A Simple GUI
   - a regular n-gon
   - recursive images

2. The Cantor Set
   - recursive definition
   - recursive drawing algorithm

3. The Koch Curve and Flake
   - recursive drawing algorithm
   - GUI for a Koch flake

4. Space Filling Curves and L-systems
   - a GUI for Hilbert curves
   - implementing the moves
def amove(self, k):
    """
    Turns counterclockwise from up right to down right.
    """
    if k > 0:
        self.dmove(k-1)
        self.plot(self.xpt - self.hgt, self.ypt)
        self.amove(k-1)
        self.plot(self.xpt, self.ypt - self.hgt)
        self.amove(k-1)
        self.plot(self.xpt + self.hgt, self.ypt)
        self.bmove(k-1)
def bmove(self, k):
    
    #
    # Turns clockwise from down left to down right.
    #
    if k > 0:
        self.cmove(k-1)
        self.plot(self.xpt, self.ypt + self.hgt)
        self.bmove(k-1)
        self.plot(self.xpt + self.hgt, self.ypt)
        self.bmove(k-1)
        self.plot(self.xpt, self.ypt - self.hgt)
        self.amove(k-1)
the Move C

C: B → C ↑ C ← B

def cmove(self, k):
    """
    Turns counterclockwise from down left to up left.
    """
    if k > 0:
        self.bmove(k-1)
        self.plot(self.xpt + self.hgt, self.ypt)
        self.cmove(k-1)
        self.plot(self.xpt, self.ypt + self.hgt)
        self.cmove(k-1)
        self.plot(self.xpt - self.hgt, self.ypt)
        self.dmove(k-1)
def dmove(self, k):
    """
    Turns clockwise from up right to up left.
    """
    if k > 0:
        self.amove(k-1)
        self.plot(self.xpt, self.ypt - self.hgt)
        self.dmove(k-1)
        self.plot(self.xpt - self.hgt, self.ypt)
        self.dmove(k-1)
        self.plot(self.xpt, self.ypt + self.hgt)
        self.cmove(k-1)
Summary

Applying simple rules recursively, we can draw remarkable pictures.

- For recursive images, the design of a GUI with canvas, scale, and button, can always be very similar. The action is defined by a callback function.
- We start with simple drawings, such as a regular n-gon or a Cantor set, before making more complicated plots.
- The more advanced mathematical computations are defined in separate functions, isolating the complexity to some critical spots.
Exercises

1. Make a GUI to visualize the Sierpinski gasket.

2. A variant of the Sierpinski gasket starts with a square and removes a smaller square at the center of the first square. This removal is repeated to the 8 remaining squares. Design a GUI for this Sierpinski carpet. Define the recursive drawing algorithm.

3. Make a GUI to visualize a Brownian bridge between two points. The rule is to replace a line segment from $A$ to $B$ by the segments $[A, M]$ and $[M, B]$, where $M$ is calculated as $(A + B)/2$ with random noise added to it. Repeat the rule to the new segments, etc.

4. Design a recursive algorithm to draw a tree, starting with a trunk and 3 branches. Put at the top of each branch again a trunk and 3 branches, and then again... What are the parameters of the recursive function to define the rule to generate the tree?