21. Recursion

What is Recursion?

A function is recursive if it calls itself.

A pattern is recursive if it is defined in terms of itself.

I can tell you what this is in terms of what that is.

The Concept of Recursion Is Hard But VERY Important

Teaching Plan:
- Develop a recursive triangle-tiling procedure informally.
- Fully implement (in Python) a recursive rectangle-tiling procedure.
- Fully implement a recursive function for n!.
- Fully implement a recursive function for sorting (in a later lecture).

Recursive Graphics

We will develop a graphics procedure that draws this:

The procedure will call itself.

Tiling a Triangle

We start with one big triangle:

And are to end up with this:
Requires Repetition

Given a yellow triangle
Define the inner triangle and the 3 corner triangles
Color the inner triangle and repeat the process on the 3 corner triangles

“Repeat the Process”

Visit every yellow triangle and replace it with this

We Get This...

“Repeat the Process”

Visit every yellow triangle and replace it with

We Get This...

“Repeat the Process”

Visit every yellow triangle and replace it with
We Get This...

The Notion of Level

A level-0 tiling
A level-1 tiling
A level-2 tiling
A level-3 tiling

The Connection Between Levels

To display a level-3 tiling you do this:
- display the inner triangle $T_0$
- display a level-2 tiling of corner triangles $T_1$, $T_2$, and $T_3$

The Connection Between Levels

To display an level-$L$ tiling you do this:
- display the inner triangle $T_0$
- display an level-$(L-1)$ tiling of triangles $T_1$, $T_2$, and $T_3$

Some Tools to Pull This Off

class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def Mid(self, other):
        """ Returns a point that encodes the midpoint of the line segment that connects the Point self and the Point other. """

Midpoints from Vertices

P1
P2
P3
P12 = P1.Mid(P2)
P23 = P2.Mid(P3)
P31 = P3.Mid(P1)
def DrawTriangle(P1, P2, P3, c):
    """ Draws a triangle with vertices
    P1, P2, and P3 and FillColor c
    PreC: P1, P2, and P3 are points
    and c is a rgb list.
    """

The Overall Procedure

    def Tile(P1, P2, P3, L):
        if L == 0:
            # Base case. Draw a yellow triangle
            DrawTriangle(P1, P2, P3, YELLOW)
        else:
            # Compute side midpoints P12, P23, P31
            # Color inner triangle MAGENTA
            # Draw level L-1 tilings of T1, T2, T3

The Overall Procedure

    # Compute side midpoints P12, P23, P31
    P12 = P1.Mid(P2)
    P23 = P2.Mid(P3)
    P31 = P3.Mid(P1)

The Overall Procedure

    # Color the inner triangle magenta
    DrawTriangle(P12, P23, P31, MAGENTA)

The Overall Procedure

    # Draw level-(L-1) tilings of T1, T2, T3
    Tile(P1, P31, P12, L-1)
    Tile(P2, P12, P23, L-1)
    Tile(P3, P23, P31, L-1)

These are the recursive calls.
The Overall Procedure

```python
def Tile(P1, P2, P3, L):
    if L == 0:
        # Base case. Draw a yellow triangle
        DrawTriangle(P1, P2, P3, YELLOW)
    else:
        # Compute side midpoints P12, P23, P31
        # Color inner triangle MAGENTA
        # Draw level L-1 tilings of T1, T2, T3
        Tile(P1, P31, P12, L-1)
        Tile(P2, P12, P23, L-1)
        Tile(P3, P23, P31, L-1)
```

A Note on Chopping up a Region into Triangles...

It is Important!

Step One in simulating flow around an airfoil is to generate a triangular mesh and (say) estimate the velocity at each little triangle using physics and math.

Next Up

A Non-Graphics Example of Recursion: The Factorial Function

Recursive Evaluation of Factorial

Recall the factorial function:

```python
def F(n):
    x = 1
    for k in range(1, n+1):
        x = x*k
    return x
```

Q. How would you compute $6!$ given that you have computed $5! = 120$?

A. $6! = 120 \times 6$
Recursive Evaluation of Factorial

How does this work?

```python
def F(n):
    if n<=1:
        return 1
    else:
        a = F(n-1)
        return n*a
```

Executing F(3)

```python
m = 3
x = F(m)
print x
```

We are in the calling script

```
def F(n):
    if n<=1:
        return 1
    else:
        a = F(n-1)
        return n*a
```

Executing F(3)

```
def F(n):
    if n<=1:
        return 1
    else:
        a = F(n-1)
        return n*a
```

The function F is called with argument 3. We open up a call frame.

```
def F(n):
    if n<=1:
        return 1
    else:
        a = F(n-1)
        return n*a
```

Executing F(3)

```
def F(n):
    if n<=1:
        return 1
    else:
        a = F(n-1)
        return n*a
```

We open up a call frame.

```
def F(n):
    if n<=1:
        return 1
    else:
        a = F(n-1)
        return n*a
```

We encounter a function call. F is called with argument equal to 2.

```
def F(n):
    if n<=1:
        return 1
    else:
        a = F(n-1)
        return n*a
```

We encounter a function call. F is called with argument equal to 1.
Executing F(3)

```
m = 3
x = F(m)
print x
```

We open up a call frame.

```
def F(n):
    if n<=1:
        return 1
    else:
        a = F(n-1)
        return n*a
```

The value of 1 is "assigned" to return.

```
m → 3
x → 3
n → 2
a → 1
return
```

```
m → 3
x → 3
n → 2
a → 1
return
```

```
m → 3
x → 3
n → 2
a → 1
return
```

That function call is over.

Control now passes to this "edition" of F.

```
m = 3
x = F(m)
print x
```

```
def F(n):
    if n<=1:
        return 1
    else:
        a = F(n-1)
        return n*a
```

Control passes to this "edition" of F. The value 2 is "assigned" to return.
Executing F(3)

\[
\begin{align*}
m &= 3 \\
x &= F(m) \\
\text{print } x
\end{align*}
\]

```python
def F(n):
    if n<=1:
        return 1
    else:
        a = F(n-1)
        return n*a
```

The value is returned to the caller.

\[
\begin{align*}
m &\rightarrow 3 \\
x &\rightarrow \\
\text{return}
\end{align*}
\]

The function call is over.

\[
\begin{align*}
m &\rightarrow 3 \\
x &\rightarrow 3 \\
a &\rightarrow 2 \\
\text{return } 2
\end{align*}
\]

Control now passes to this "edition" of F.

\[
\begin{align*}
m &\rightarrow 3 \\
x &\rightarrow \\
\text{return}
\end{align*}
\]

The value is returned to the caller.

\[
\begin{align*}
m &\rightarrow 3 \\
x &\rightarrow 6 \\
a &\rightarrow 2 \\
\text{return } 6
\end{align*}
\]

This function call is over.
Executing F(3)

\[
\begin{align*}
  m & = 3 \\
  x & = F(m) \\
  \text{print } x \\
\end{align*}
\]

\[
\begin{align*}
  m & \rightarrow 3 \\
  x & \rightarrow 6 \\
\end{align*}
\]

Control passes to the script that asked for F(3)

Output: 6

All Done!

Another Example: Random Mondrians

Using Python:

Random Mondrian

Given This:

Random Mondrian

Draw This:

The Subdivide Process Applies to a Rectangle

Given a rectangle specified by its length, width, and center, either randomly color it or randomly subdivide it.
Subdivision Starts with a Random Dart Throw

This Defines 4 Smaller Rectangles

This Defines 4 Smaller Rectangles

The Notion of Level

A 1-level Partitioning  A 2-level Partitioning

We can again repeat the process on each of the 16 smaller rectangles. Etc.

Pseudocode

```python
def Mondrian(x,y,L,W,level):
    if level == 0:
        c = RandomColor()
        DrawRect(x,y,L,W,FillColor=c)
    else:
        # Subdivide into 4 smaller rectangles
        Mondrian(upper left rectangle info, level-1)
        Mondrian(upper right rectangle info, level-1)
        Mondrian(lower left rectangle info, level-1)
        Mondrian(lower right rectangle info, level-1)
```

How to Generate Random Colors

We need some new technology to organize the selection random colors.

We need lists whose entries are lists.
Lists with Entries that Are Lists

An Example:

cyan = [0.0,1.0,1.0]
magenta = [1.0,0.0,1.0]
yellow = [1.0,1.0,0.0]
colorList = [cyan,magenta,yellow]

Pick a Color at Random

cyan = [0.0,1.0,1.0]
magenta = [1.0,0.0,1.0]
yellow = [1.0,1.0,0.0]
colorList = [cyan,magenta,yellow]
r = randi(0,2)
randomColor = colorList[r]

Package the Idea...

from simpleGraphics import *
from random import randint as randi
def RandomColor():
    """ Returns a randomly selected rgb list."""
    c = [RED, GREEN, BLUE, ORANGE, CYAN]
    i = randi(0,len(c)-1)
    return c[i]

How to Randomly Subdivide a Rectangle

L
(x,y)  (x,y)
W
xc = randu(x-L/2,x+L/2)
yc = randu(y-W/2,y+W/2)

The Math Behind the Little Rectangles

The upper right rectangle is typical:

Length: \( L_1 = (x+L/2) - xc \)
Width: \( W_1 = (y+W/2) - yc \)
Center: \( (xc+L_1/2, yc+W_1/2) \)

The Procedure Mondrian

A couple of features to make the design more interesting:

1. The dart throw that determines the subdivision can’t land too near the edge. No super skinny tiles!
2. Randomly decide whether or not to subdivide. This creates a nice diversity in size.
Overall Conclusions

Recursion is sometimes the simplest way to organize a computation.

It would be next to impossible to do the triangle tiling problem any other way.

On the other hand, factorial computation is easier via for-loop iteration.

Overall Conclusions

Infinite recursion (like infinite loops) can happen so careful reasoning is required.

Will we reach the "base case"?

Graphics examples: We will reach Level==0
Factorial: We will reach n==1