17. Recursion

What is Recursion?
A function is recursive if it calls itself.

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

Recursive Graphics
We will develop a graphics procedure that draws this:

The procedure will call itself.

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).

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

The Connection Between Levels

The Connection Between Levels

A Recursive Procedure

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.

Another Example: Random Mondrians

Using Python:

Random Mondrian

Given This:

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

Repeat the process on each of the 4 smaller rectangles...

This Defines 4 Smaller Rectangles

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

The Notion of Level

A 1-level Partitioning

A 2-level Partitioning

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)
```

We look at a few details. Complete implementation online.

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:

```python
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

\[ \begin{align*}
(xc,yc) & \quad (x,y) \\
L & \quad W
\end{align*} \]

\[
xc = \text{randu}(x-L/2,x+L/2) \\
yc = \text{randu}(y-W/2,y+W/2)
\]

The Math Behind the Little Rectangles

\[ \begin{align*}
(xc,yc) & \quad (x,y) \\
L & \quad W
\end{align*} \]

The upper right rectangle is typical:

| Length: | \( L_1 = (x+L/2) - xc \) |
| Width: | \( W_1 = (y+W/2) - yc \) |
| Center: | \((xc+L1/2,yc+W1/2)\) |

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.

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
```

5! = 1 \times 2 \times 3 \times 4 \times 5

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)

We are in the calling script

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

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

The value of 1 is "assigned" to return

The value is sent back to the caller.

That function call is over

Executing F(3)

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

```
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 1

The value is sent back to the caller.

That function call is over

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

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

We open up a call frame.

The value of 1 is "assigned" to return

The value is sent back to the caller.

That function call is over
Executing $F(3)$

$m = 3$
$x = F(m)$

print $x$

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

Control now passes to this "edition" of $F$.

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

$n --> 2$
$a --> 1$
return

The value is returned to the caller.

The function call is over.

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

$m --> 3$
$n --> 3$
$a --> 2$
return

The value 6 is "assigned" to return.
Executing F(3)

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

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

The value is returned to the caller.

```
m --&gt; 3
x --&gt; 6
a --&gt; 3
a --&gt; 2
return 6
```

This function call is over.

```
m --&gt; 3
x --&gt; 6
a --&gt; 2
return 6
```

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

Output: 6

All Done!