Turtle Graphics

Python has a built-in module named turtle. See the Python turtle module API for details.

Use `from turtle import *` to use these commands:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd(dist)</code></td>
<td>turtle moves forward by <code>dist</code></td>
</tr>
<tr>
<td><code>bk(dist)</code></td>
<td>turtle moves backward by <code>dist</code></td>
</tr>
<tr>
<td><code>lt(angle)</code></td>
<td>turtle turns left <code>angle</code> degrees</td>
</tr>
<tr>
<td><code>rt(angle)</code></td>
<td>turtle turns right <code>angle</code> degrees</td>
</tr>
<tr>
<td><code>pu()</code></td>
<td>(pen up) turtle raises pen in belly</td>
</tr>
<tr>
<td><code>pd()</code></td>
<td>(pen down) turtle lower pen in belly</td>
</tr>
<tr>
<td><code>pensize(width)</code></td>
<td>sets the thickness of turtle's pen to <code>width</code></td>
</tr>
<tr>
<td><code>pencolor(color)</code></td>
<td>sets the color of turtle's pen to <code>color</code></td>
</tr>
<tr>
<td><code>shape(shp)</code></td>
<td>sets the turtle's shape to <code>shp</code></td>
</tr>
<tr>
<td><code>home()</code></td>
<td>turtle returns to (0,0) (center of screen)</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>delete turtle drawings; no change to turtle's state</td>
</tr>
<tr>
<td><code>reset()</code></td>
<td>delete turtle drawings; reset turtle's state</td>
</tr>
<tr>
<td><code>setup(width, height)</code></td>
<td>create a turtle window of given <code>width</code> and <code>height</code></td>
</tr>
</tbody>
</table>

A Simple Example with Turtles

The only two commands that draw lines are `fd` and `bk`.

```python
from turtle import *
setup(400, 400)
fd(100)
lt(60)
shape('turtle')
pencolor('red')
fd(150)
rt(15)
pencolor('blue')
bk(100)
pu()
bk(50)
pd()
pensize(5)
bk(250)
pensize(1)
home()
exitonclick()
```

Loopying Turtles (1)

Loops can be used in conjunction with turtles to make interesting designs.

```python
def polygon(numSides, sideLength):
    """ Draws a polygon with the specified number of sides, each with the specified length. """
```

Will solve this in the Notebook.
**Looping Turtles (2)**

```python
def polyFlow(numPetals, petalSides, petalLen):
    """Draws 'flowers' with numPetals arranged around a center point. Each petal is a polygon with petalSides sides of length petalLen."
    """
```

Will solve this in the Notebook.

```
spiral(200, 90, 0.9, 10)
spiral(200, 72, 0.97, 10)
spiral(200, 80, 0.95, 10)
spiral(200, 95, 0.93, 10)
spiral(200, 121, 0.95, 15)
```

**Spiraling Turtles: A Recursion Example**

**Answer this:**
How would you create these shapes using loops? Recursion makes easier solving certain problems that involve a repeating pattern.

```
def spiral(sideLen, angle, scaleFactor, minLength):
    """Draw a spiral recursively."""
    if sideLen >= minLength:
        fd(sideLen)
        lt(angle)
        spiral(sideLen*scaleFactor, angle, scaleFactor, minLength)
```

<table>
<thead>
<tr>
<th>SideLen</th>
<th>Angle</th>
<th>Scale Factor</th>
<th>Min Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>625</td>
<td>90</td>
<td>0.8</td>
<td>250</td>
</tr>
</tbody>
</table>

**Concepts in this slide:**
Drawing function call frames helps us follow the execution of recursion.

```
spiral(625, 90, 0.8, 250)
```
### Turtle Recursion

Initially all execution frames co-exist in the memory. Only once a function has returned (implicitly), the execution frame is deleted.

---

### Invariant Spiraling

A function is invariant relative to an object’s state if the state of the object is the same before and after the function is invoked.

```python
def spiralBack(sideLen, angle, scaleFactor, minLength):
    """ Draws a spiral. The state of the turtle (position, color, heading, etc.) after drawing the spiral is the same as before drawing the spiral. ""
    if sideLen >= minLength:
        fd(sideLen)
        lt(angle)
        spiralBack(sideLen*scaleFactor, angle, scaleFactor, minLength)
        rt(angle)
        bk(sideLen)
```

---

**Important**
All execution frames were one by one deleted after their completion. This terminates the invocation of the function and has created as a “side-effect” the turtle image at the top of the slide.
How does spiralBack work?

spiralBack(625, 90, 0.8, 250)

```python
if True:
    fd(625)
    lt(90)
    spiralBack(500, 90, 0.8, 250)
    rt(angle)
    bk(sideLen)
spiralBack(500, 90, 0.8, 250)
    if True:
        fd(500)
        lt(90)
        spiralBack(400, 90, 0.8, 250)
        rt(angle)
        bk(sideLen)
    spiralBack(400, 90, 0.8, 250)
        if True:
            fd(400)
            lt(90)
            spiralBack(320, 90, 0.8, 250)
            rt(angle)
            bk(sideLen)
            spiralBack(320, 90, 0.8, 250)
                if True:
                    fd(320)
                    lt(90)
                    spiralBack(256, 90, 0.8, 250)
                    rt(angle)
                    bk(sideLen)
                spiralBack(256, 90, 0.8, 250)
                    if True:
                        fd(256)
                        lt(90)
                        spiralBack(204.8, 90, 0.8, 250)
                        rt(angle)
                        bk(sideLen)
                spiralBack(204.8, 90, 0.8, 250)
                    if True:
                        fd(204.8)
                        lt(90)
                        spiralBack(153.6, 90, 0.8, 250)
                        rt(angle)
                        bk(sideLen)
                spiralBack(153.6, 90, 0.8, 250)
```

spiralBack(625, 90, 0.8, 250)

```
if False:
    fd(sideLen)
    lt(angle)
    spiralBack(sideLen*scaleFactor,...)
    rt(angle)
    bk(sideLen)
```

Turtle Recursion 19-13

Essence of Invariance

Do state change 1
Do state change 2
Do state change n-1
Do state change n

Recursive call to function

Undo state change n
Undo state change n-1
...
Undo state change 2
Undo state change 1

Perform changes to state

Undo state changes in opposite order

Turtle Recursion 19-14

Trees

tree(7, 75, 30, 0.8)
tree(10, 75, 15, 0.8)
tree(10, 80, 45, 0.7)
tree(10, 100, 90, 0.68)

Draw a tree recursively

```python
tree(levels, trunkLen, angle, shrinkFactor)
```

- **levels** is the number of branches on any path from the root to a leaf
- **trunkLen** is the length of the base trunk of the tree
- **angle** is the angle from the trunk for each subtree
- **shrinkFactor** is the shrinking factor for each subtree

Turtle Recursion 19-15

Turtle Recursion 19-16
How to make a 4-level tree:
\[ \text{tree}(4, 100, 45, 0.6) \]

1. Make a trunk of size 100
2. Make two 3-level trees with 60% trunks set at 45° angles

How to make a 3-level tree:
\[ \text{tree}(3, 60, 45, 0.6) \]

1. Make a trunk of size 60
2. Make two 2-level trees with 60% trunks set at 45° angles

How to make a 2-level tree:
\[ \text{tree}(2, 36, 45, 0.6) \]

1. Make a trunk of size 36
2. Make two 1-level trees set at 45° angles
3. Do nothing!

How to make a 1-level tree:
\[ \text{tree}(1, 21.6, 45, 0.6) \]

1. Make a trunk of size 21.6
2. Make two 0-level trees set at 45° angles

How to make a 0-level tree:
\[ \text{tree}(0, 12.96, 45, 0.6) \]

1. Do nothing!

---

def tree(levels, trunkLen, angle, shrinkFactor):
    
    """Draw a 2-branch tree recursively.
    
    levels: number of branches on any path from the root to a leaf
    trunkLen: length of the base trunk of the tree
    angle: angle from the trunk for each subtree
    shrinkFactor: shrinking factor for each subtree
    """
    if levels > 0:
        # Draw the trunk.
        fd(trunkLen)
        # Turn and draw the right subtree.
        rt(angle)
        tree(levels-1, trunkLen*shrinkFactor, angle, shrinkFactor)
        # Turn and draw the left subtree.
        lt(angle * 2)
        tree(levels-1, trunkLen*shrinkFactor, angle, shrinkFactor)
        # Turn back and back up to root without drawing.
        rt(angle)
        pu()
        bk(trunkLen)
        pd()
The squirrels aren't fooled...

```
def treeRandom(length, minLength, thickness, minThickness, minAngle, maxAngle, minShrink, maxShrink):
    if (length < minLength) or (thickness < minThickness): # Base case
        pass # Do nothing
    else:
        angle1 = random.uniform(minAngle, maxAngle)
        angle2 = random.uniform(minAngle, maxAngle)
        shrink1 = random.uniform(minShrink, maxShrink)
        shrink2 = random.uniform(minShrink, maxShrink)
        pensize(thickness)
        fd(length)
        rt(angle1)
        treeRandom(length*shrink1, minLength, thickness*shrink1, minThickness, minAngle, maxAngle, minShrink, maxShrink)
        lt(angle1 + angle2)
        treeRandom(length*shrink2, minLength, thickness*shrink2, minThickness, minAngle, maxAngle, minShrink, maxShrink)
        rt(angle2)
        pensize(thickness)
        bk(length)
```

More resources

Full Slides: http://cs111.wellesley.edu/lectures/lecture19

- All steps of recursion examples, drawn out.
- Exercises for drawing Koch curves and snowflakes with recursive turtle functions.
- History about turtles at Wellesley and elsewhere.
- Applying the turtle programming abstraction to control laser cutters in the WeLab (Wellesley engineering lab).

Random Trees

Laser Cutting a Tree with Turtle Blocks