CS134:
Graphical Recursion

Announcements & Logistics

- **Lab 7** will be posted today: focuses on recursion
  - Please **complete Task 0** before you come to lab!!!
  - Quick note about **command line arguments**

    ```
    >>> python3 bedtime.py duck cow dog
    ```

  - Interpreted as a list of strings called `argv`; we provide the code for you in starter
- **HW 6** due Monday @ 10 pm: covers sorting, dictionaries, sets, tuples
- **Scheduled final:** Fri, Dec 16, 9:30 am, details TBD

- **CS TA applications** due October 28 (**today**!)
  - Feel free to submit Iris, Jeannie, Lida as references
  - https://csci.williams.edu/tatutor-application/

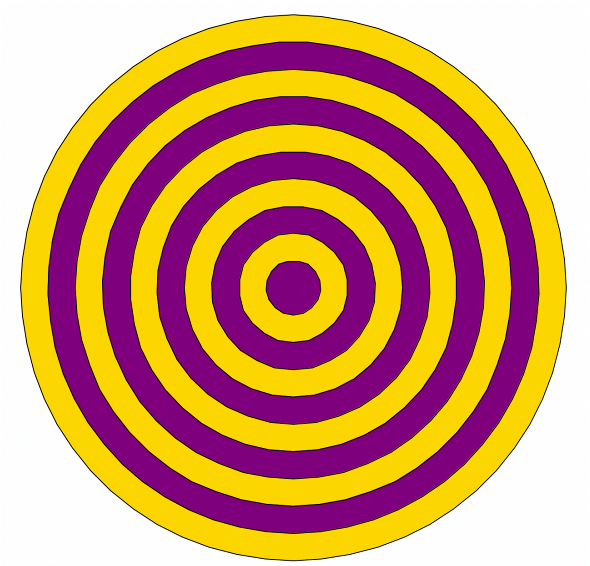
Do You Have Any Questions?

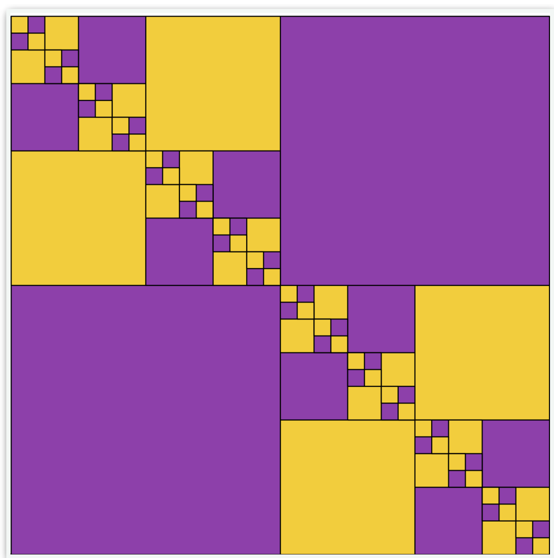
Last Time: Recursive Approach to Problem Solving

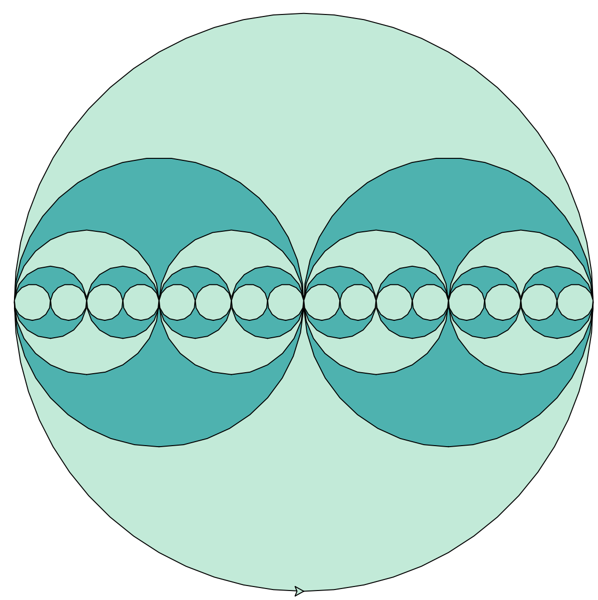
- A recursive function is a function **that calls itself**
- A recursive approach to problem solving has two main parts:
  - **Base case(s).** When the problem is **so small**, we solve it directly, without having to reduce it any further
  - **Recursive step.** Does the following things:
    - Performs an action that contributes to the solution
    - **Reduces** the problem to a smaller version of the same problem, and calls the function on this **smaller subproblem**
- The recursive step is a form of "wishful thinking’’ (also called the inductive hypothesis)

Today's Plan

- Introduction to Turtle
- Graphical recursion examples
- Understanding function **invariance** and why it matters when doing recursion







The Turtle Module

- Turtle is a **graphics module** first introduced in the 1960s by computer scientists Seymour Papert, Wally Feurzig, and Cynthia Solomon.
- It uses a programmable cursor — fondly referred to as the "turtle" — to draw on a Cartesian plane (x and y axis.)

Turtle In Python

- `turtle` is available as a built-in module in Python. See the Python turtle module API for details.
- Basic turtle commands:

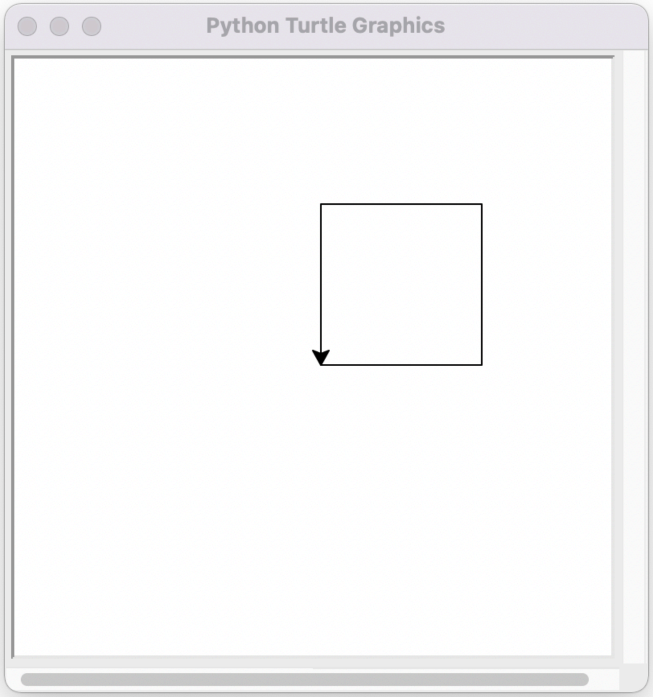
Use **`from turtle import *`** to use these commands:

`fd(`*`dist`*`)`	turtle moves forward by *dist*
`bk(`*`dist`*`)`	turtle moves backward by *dist*
`lt(`*`angle`*`)`	turtle turns left *angle* degrees
`rt(`*`angle`*`)`	turtle turns right *angle* degrees
`up()`	(pen up) turtle raises pen in belly
`down()`	(pen down) turtle lower pen in belly
`pensize(`*`width`*`)`	sets the thickness of turtle's pen to *width*
`pencolor(`*`color`*`)`	sets the color of turtle's pen to *color*
`shape(`*`shp`*`)`	sets the turtle's shape to *shp*
`home()`	turtle returns to (0,0) (center of screen)
`clear()`	delete turtle drawings; no change to turtle's state
`reset()`	delete turtle drawings; reset turtle's state
`setup(`*`width`*`,`*`height`*`)`	create a turtle window of given *width* and *height*

Basic Turtle Movement

- `forward(dist)` or `fd(dist)`, `left(angle)` or `lt(angle)`, `right(angle)` or `rt(angle)`, `backward(dist)` or `bk(dist)`

```
# set up a 400x400 turtle window
setup(400, 400)
reset()
fd(100)  # move the turtle forward 100 pixels

lt(90)   # turn the turtle 90 degrees to the left
fd(100)  # move forward another 100 pixels

# complete a square
lt(90)
fd(100)
lt(90)
fd(100)
done()
```

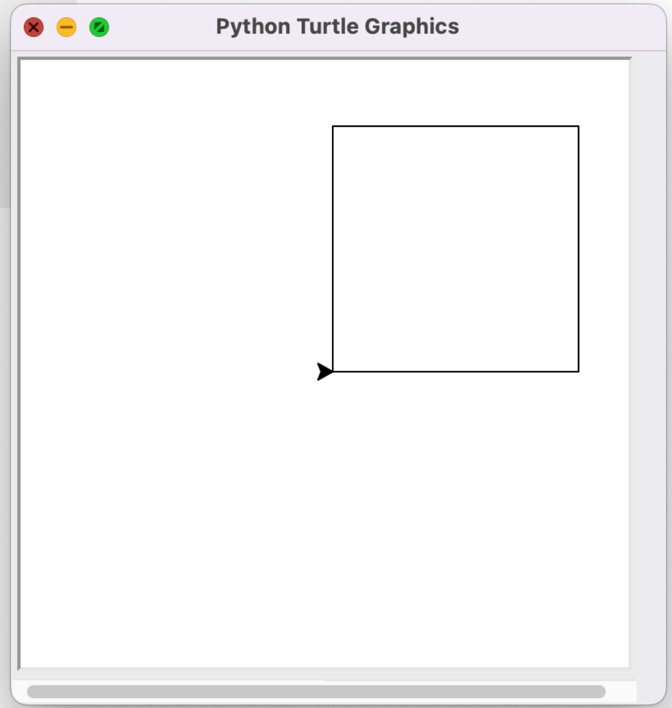


Drawing Basic Shapes With Turtle

- We can write functions that use turtle commands to draw shapes.
- For example, here's a function that draws a square of the desired size

```
def drawSquare(length):
    # a loop that runs 4 times
    # and draws each side of the square
    for i in range(4):
        fd(length)
        lt(90)
```

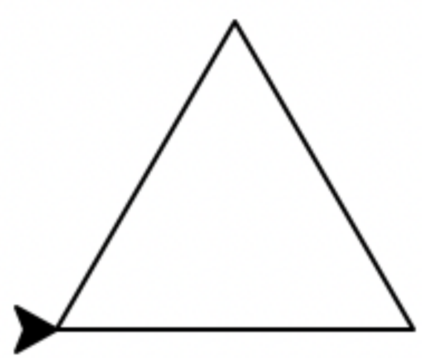
```
setup(400, 400)
reset()
drawSquare(150)
```



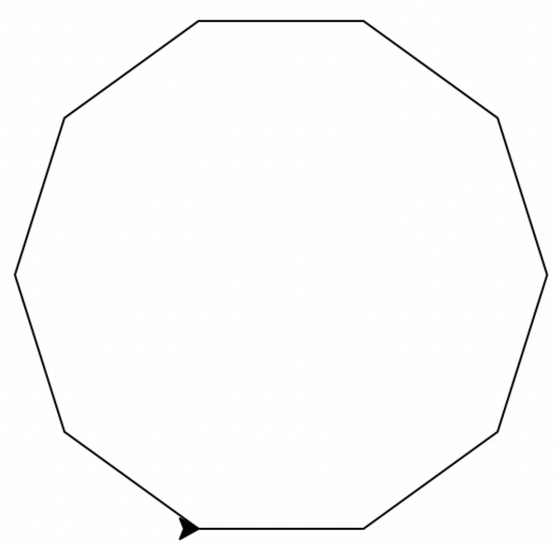
Drawing Basic Shapes With Turtle

- How about drawing polygons?

```
def drawPolygon(length, numSides):
    for i in range(numSides):
        fd(length)
        lt(360/numSides)
```



`drawPolygon(80, 3)`

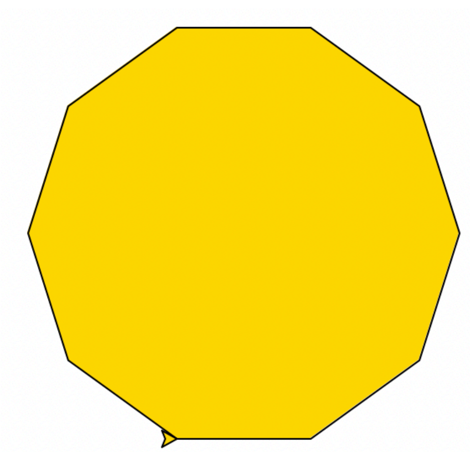


`drawPolygon(80, 10)`

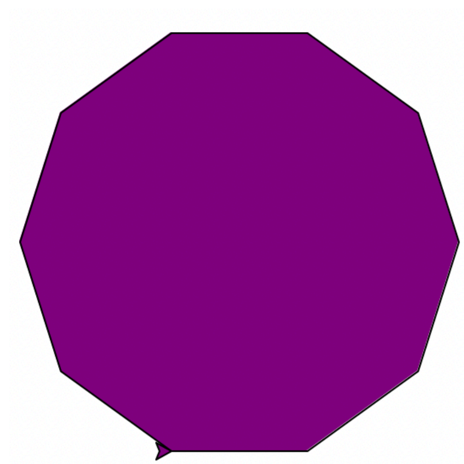
Adding Color!

- What if we wanted to add some color to our shapes?

```
def drawPolygonColor(length, numSides, color):
    # set the color we want to fill the shape with
    # color is a string
    fillcolor(color)

    begin_fill()
    for i in range(numSides):
        fd(length)
        lt(360/numSides)
    end_fill()
```



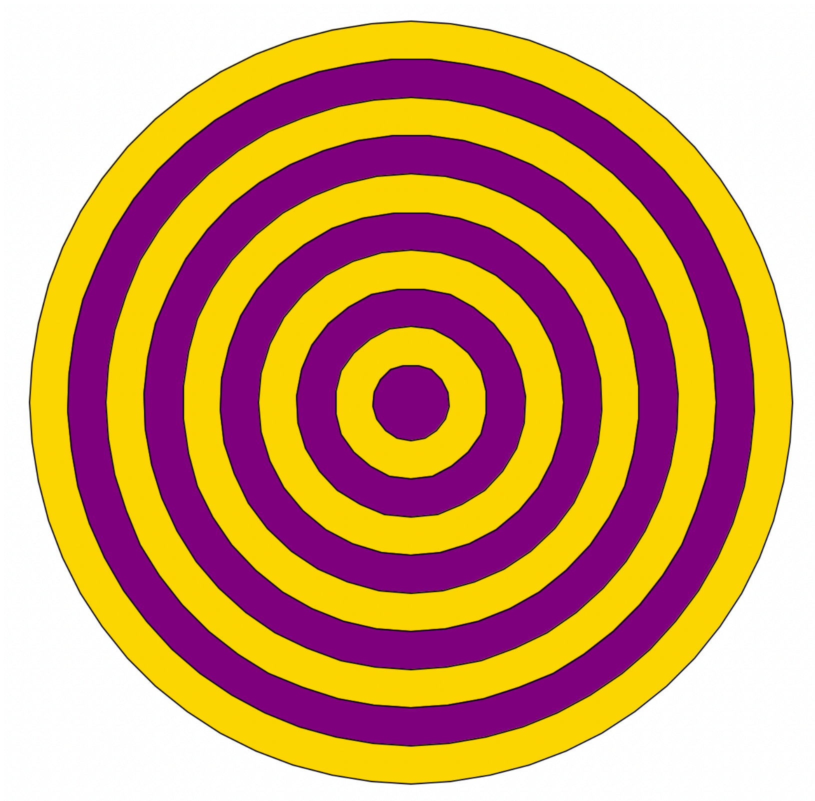
```
drawPolygonColor(80, 10, "gold")
```



```
drawPolygonColor(80, 10, "purple")
```

Recursive Figures With Turtle

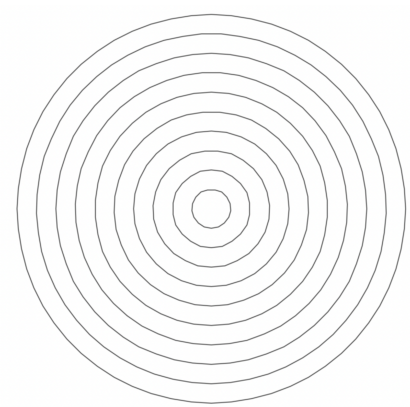
- Let's explore how to draw pretty recursive pictures with Turtle
- We'll start with figures that only require recursive calls
- Below we have a set of concentric circles of alternating colors
- How is this recursive?



Example:
Concentric Circles

Concentric Circles With No Colors

- **Recursive idea**: we have circles within circles, and each circle becomes successively smaller. In addition to drawing the circles, let's keep track of the **number of circles** we draw.
- Let's first think about the circles without colors.
- **Base case**: radius of the circle is so small it's not worth drawing, return 0
- **Recursive step**:
  - Draw a single circle of radius `r`, increment total by 1
  - Recursively draw concentric circles starting with an outer circle of a slightly smaller radius `r-g` (where `g` is any positive number you want to shrink the radius by, or the "gap" between the circles)

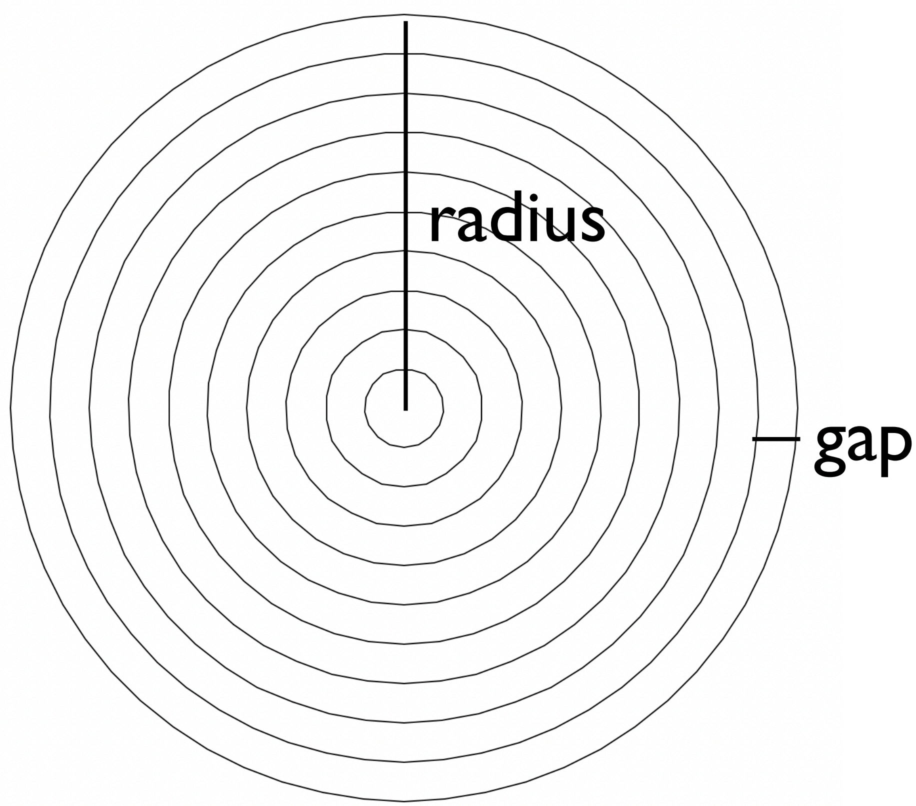


Concentric Circles

- Function definition

  `concentricCircles(radius, gap)`

  - `radius`: radius of the outermost circle
  - `gap`: width of gap between circles



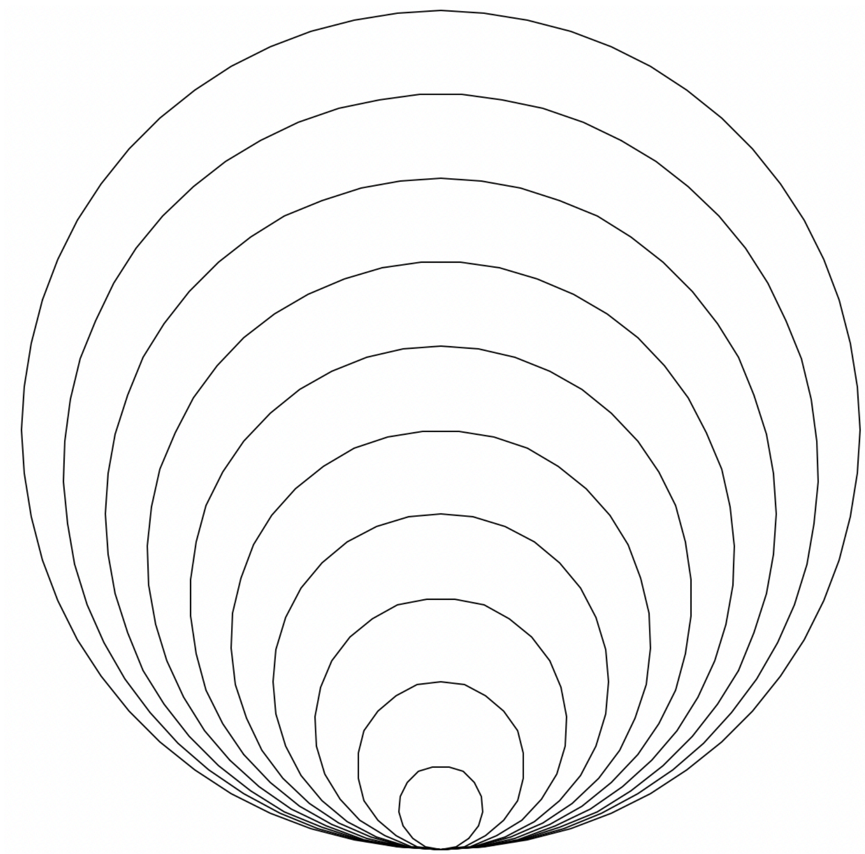
Concentric Circles

```python
def concentricCircles(radius, gap):
    # base case, don't draw anything, return 0
    if radius < gap:
        return 0
    else:
        # tell the turtle draw a circle
        circle(radius)

        # recursive function call; draw smaller circles
        num = concentricCircles(radius-gap, gap)

        # we drew one circle in this step, plus however many we
        # drew recursively, so return 1 + num
        return 1 + num
```

- Are we done?

Concentric Circles

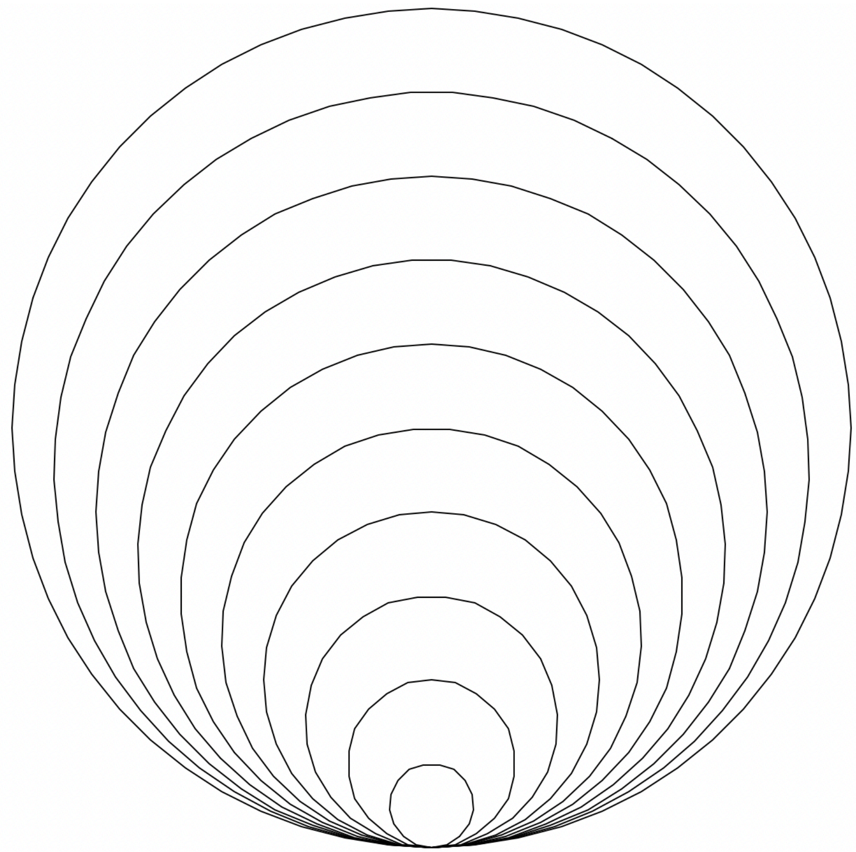
```
concentricCircles(300, 30)
```



- Pretty picture, and almost there! But not quite right. What happened?

Concentric Circles

```
concentricCircles(300, 30)
```



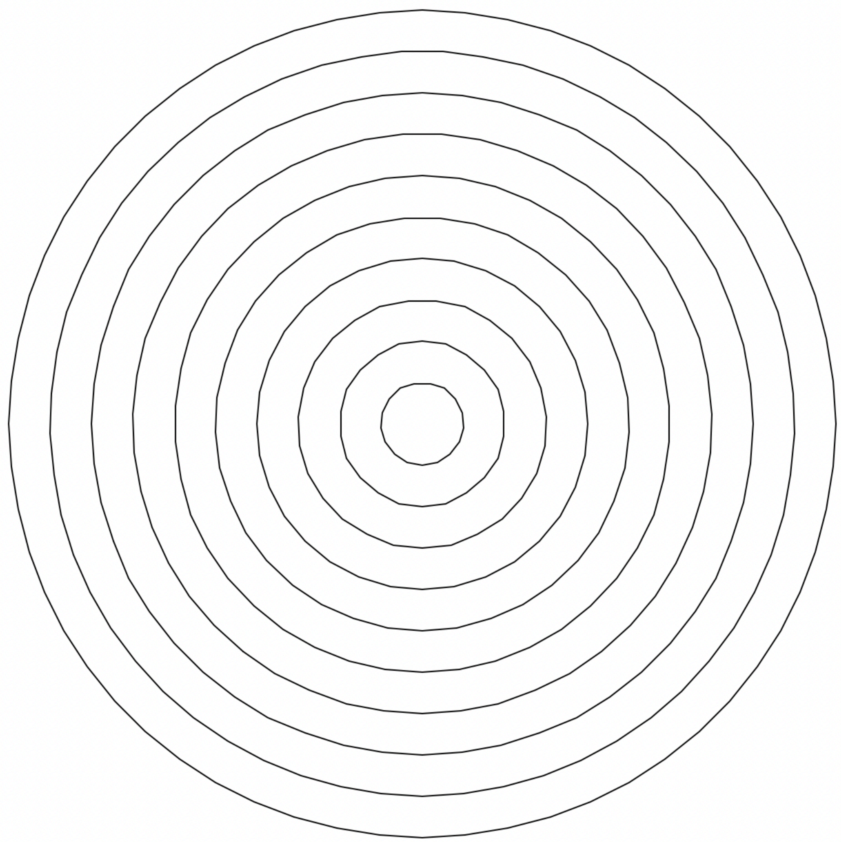
- We need to reposition the turtle after each recursive call.

Concentric Circles

```python
def concentricCircles(radius, gap):
    # base case, don't draw anything, return 0
    if radius < gap:
        return 0
    else:
        # pen down, draw circle
        down()
        circle(radius)

        # pen up, ensure the turtle doesn't draw while repositioning
        up()

        # reposition the turtle for the next circle
        lt(90)
        fd(gap)
        rt(90)

        # recursive function call; draw smaller circles
        num = concentricCircles(radius-gap, gap)

        # we drew one circle in this step, plus however many we
        # drew recursively, so return 1 + num
        return 1 + num
```

Concentric Circles

```
concentricCircles(300, 30)
```



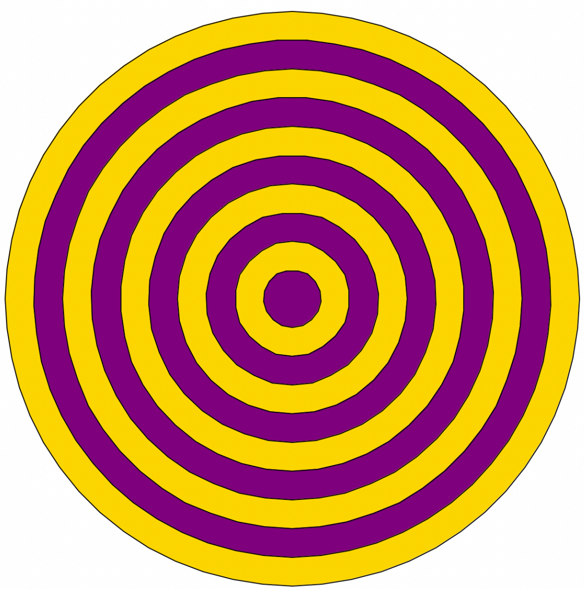
- Great! Now let's add some color.

Concentric Circles With Colors

- Function definition

```
concentricCircles(radius, gap, colorOuter, colorInner)
```

  - `radius`: radius of the outermost circle
  - `gap`: width of the gap between circles
  - `colorOuter`: color of the outermost circle
  - `colorInner`: color that alternates with colorOuter

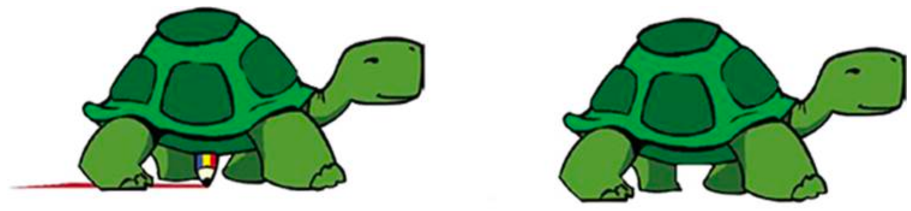


Concentric Circles: Adding Color

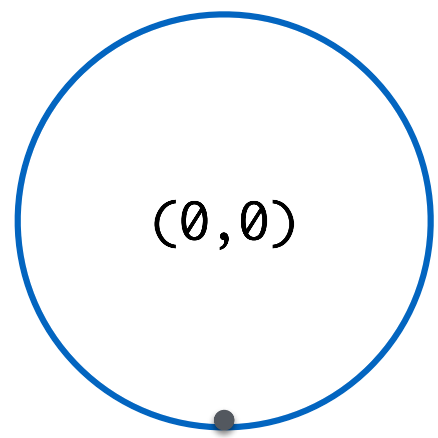
- Base case and recursive case stay the same
- How do we achieve the alternating colors?
- Just swap the order in the recursive call
  - `colorOuter` becomes `colorInner` and vice versa
- Let's also write a helper function to draw a circle filled in with some color to clean up the recursive function itself

Helper Function

```python
def drawDisc(radius, color):
    """
    Draw circle of a given radius
    and fill it with a given color
    """

    # put the pen down
    down()

    # set the color
    fillcolor(color)

    # draw the circle
    begin_fill()
    circle(radius)
    end_fill()

    # put the pen up
    up()
```



Turtle.PenDown()   Turtle.PenUp()



Starting position of turtle

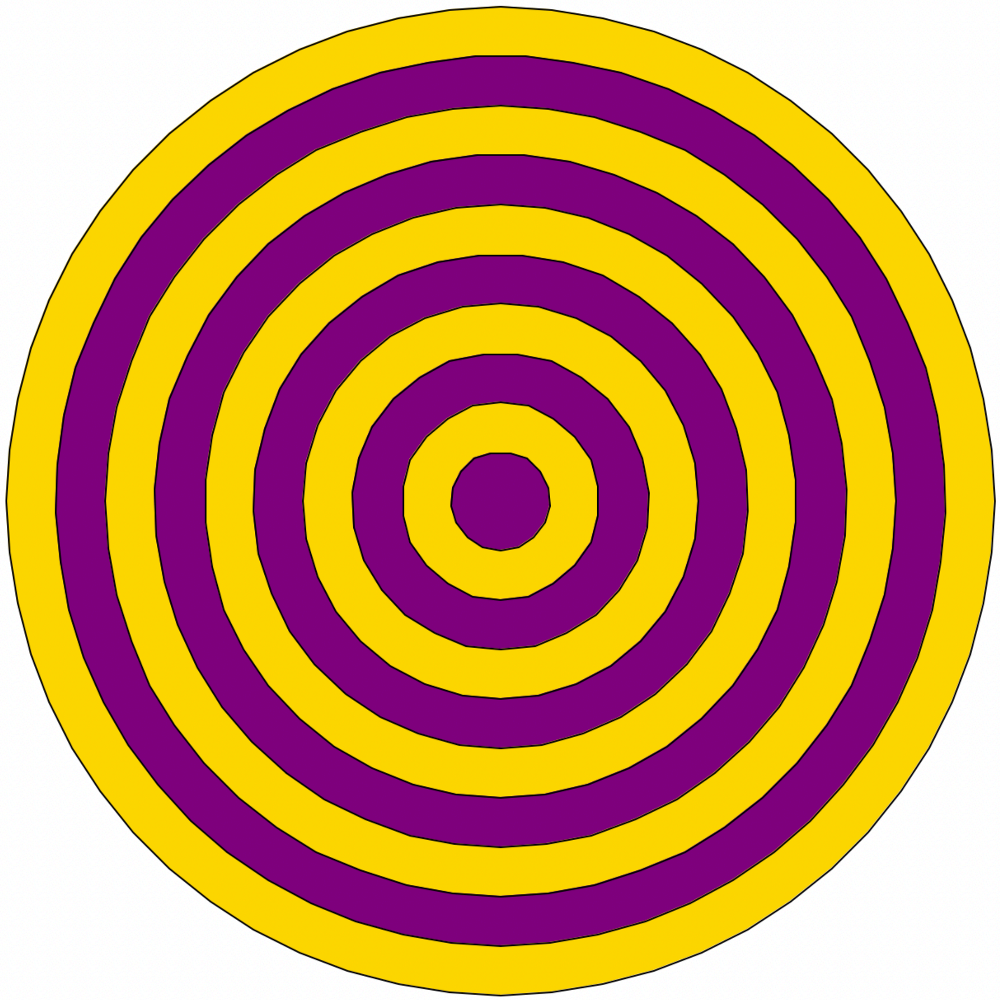
(0, -radius)

The Recursive Function

```
def concentricCirclesColor(radius, gap, colorOuter, colorInner):
    """
    Recursive function to draw concentric circles with
    alternating colors
    """
    # base case, don't draw anything, return 0
    if radius < gap:
        return 0
    else:
        drawDisc(radius, colorOuter)
        lt(90)
        fd(gap)
        rt(90)
        num = concentricCirclesColor(radius-gap, gap, colorInner, colorOuter)
        return 1 + num
```

Concentric Circles

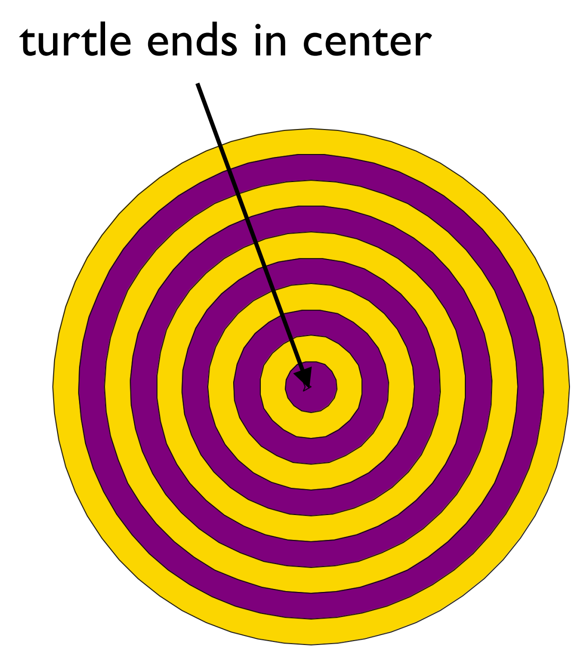
```
concentricCirclesColor(300, 30, "gold", "purple"))
```



Invariance of Functions

- A function is **invariant** if the state of the object is the same *before* and *after* the function is invoked
- Right now our `concentricCirclesColor` function is not invariant with respect to the position of the turtle
  - That is, the turtle does not end were it starts
- How can we make it invariant by returning the turtle to starting position?

```
def concentricCirclesColor(radius, gap, colorOuter, colorInner):
    """
    Recursive function to draw concentric circles with
    alternating colors
    """
    # base case, don't draw anything, return 0
    if radius < gap:
        return 0
    else:
        drawDisc(radius, colorOuter)
        lt(90)
        fd(gap)
        rt(90)
        num = concentricCirclesColor(radius-gap, gap, colorInner, colorOuter)
        return 1 + num
```



Invariant Concentric Circles

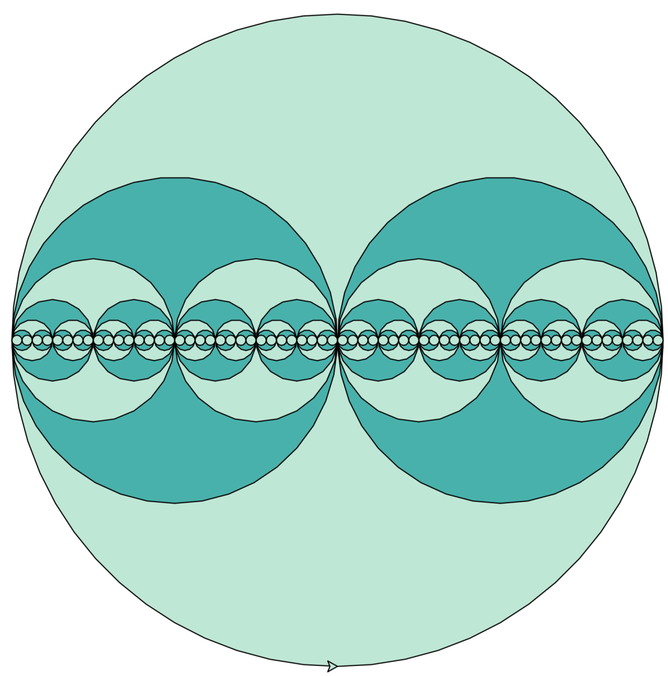
- Any turtle movements that happen before the recursive call should be "undone" after the recursive call to maintain proper invariance
- Rule of thumb: always return turtle to its starting position

```
def concentricCirclesInvariant(radius, gap, colorOuter, colorInner):
    """
    Recursive function to draw concentric circles with alternating
    color
    """
    # base case, don't draw anything, return 0
    if radius < gap:
        return 0
    else:
        drawDisc(radius, colorOuter)
        lt(90)
        fd(gap)
        rt(90)
        num = concentricCirclesInvariant(radius-gap, gap, colorInner, colorOuter)
        # move turtle back to starting position
        lt(90)
        bk(gap)
        rt(90)

        return 1 + num
```

Example: Nested Circles

Invariance of Recursive Functions

- Why do we care about **invariance**?
    - Though not always necessary for correctness, it is a good property to maintain in recursive functions
    - Our graphical functions will not always work properly if it they are not invariant
- Let's do an example with multiple recursive calls: nested circles

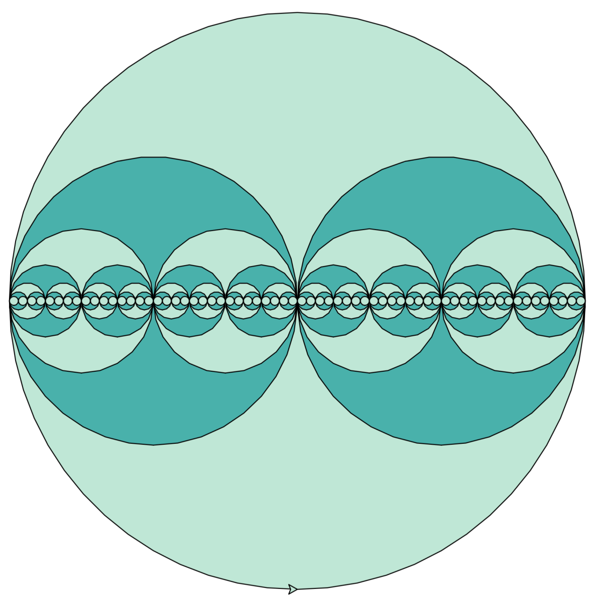


Multiple Recursive Calls

- **Example:** Nested circles function definition

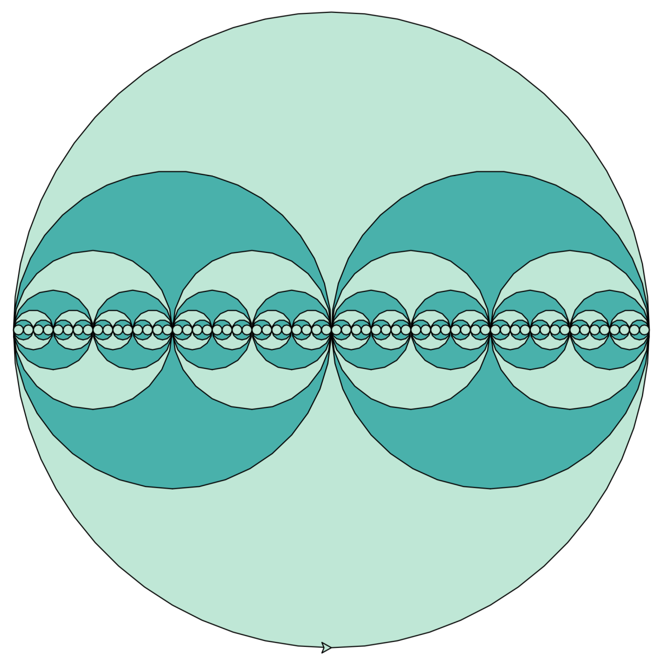
  `nestedCircles(radius, minRadius, colorOut, colorAlt)`

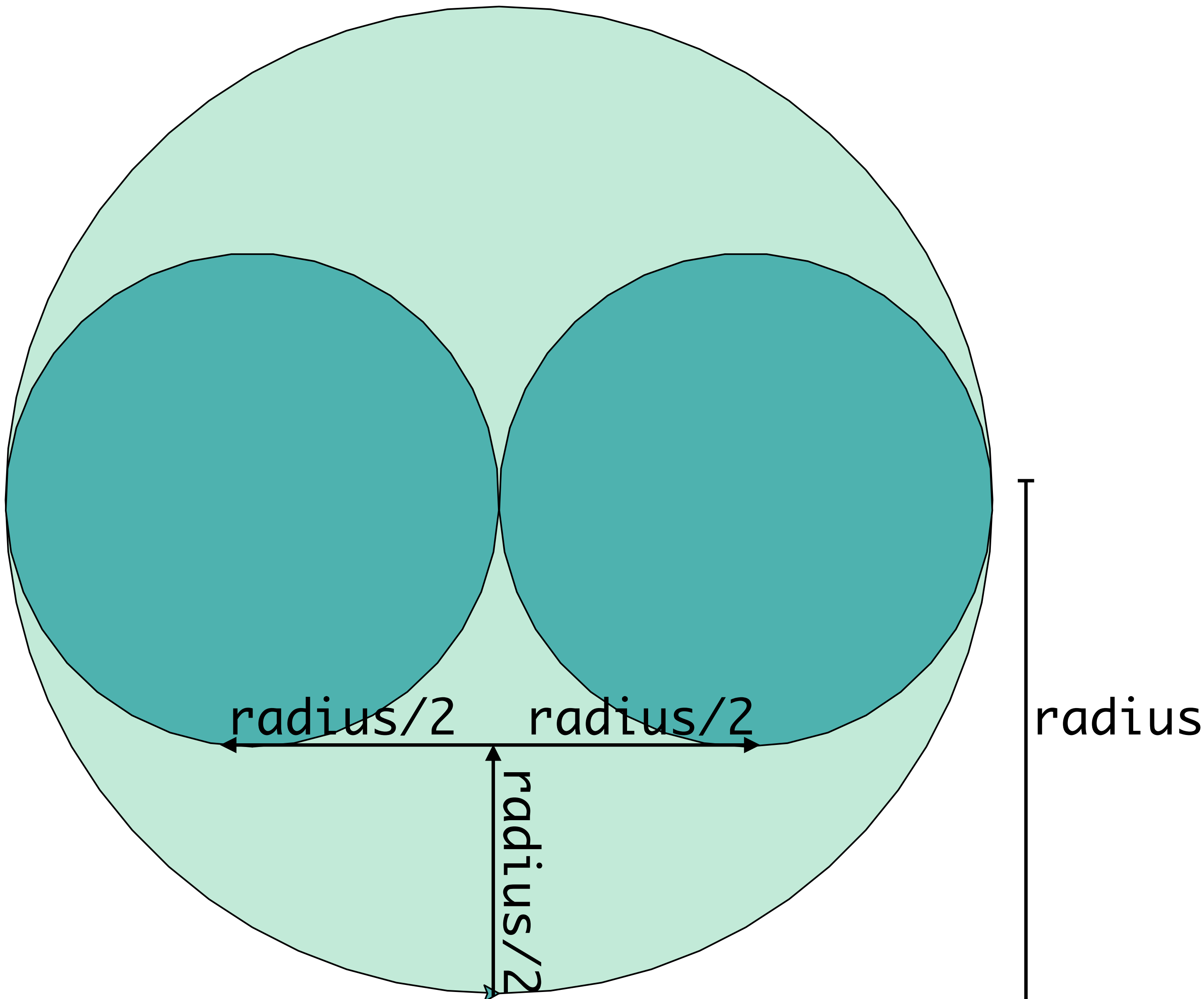
  - `radius`: radius of the outermost circle
  - `minRadius`: minimum radius of any circle
  - `colorOut`: color of the outermost circle
  - `colorAlt`: color that alternates with colorOut



Nested Circles

- **Base case?**
  - When radius becomes less than minRadius
  - Don’t draw anything return 0
- **Recursive case**
  - Draw the outer circle, add one to total
  - Position turtle for recursive calls
  - How many recursive calls do we need?



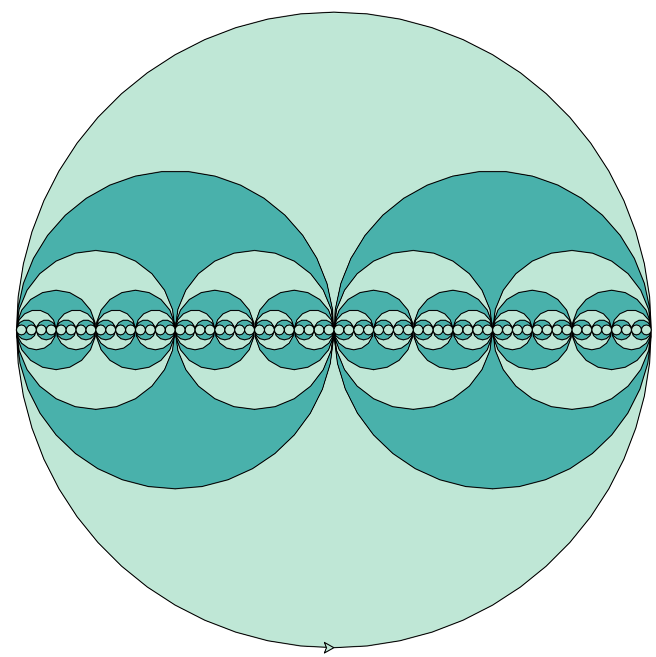


Starting position of turtle

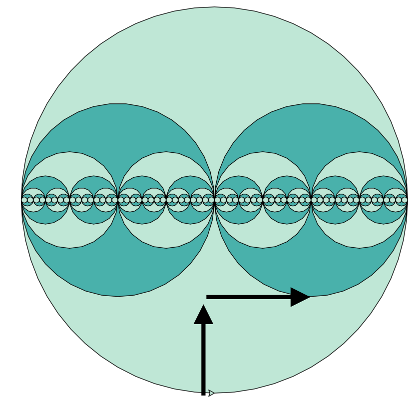
```
nestedCircles(300, 150)
```

Nested Circles

- **Base case?**
    - When radius becomes less than minRadius
    - Don't draw anything return 0
- **Recursive case**
    - Draw the outer circle, add one to total
    - Position turtle for recursive calls
    - How many recursive calls do we need?
        - Two! Right subcircle and left subcircle



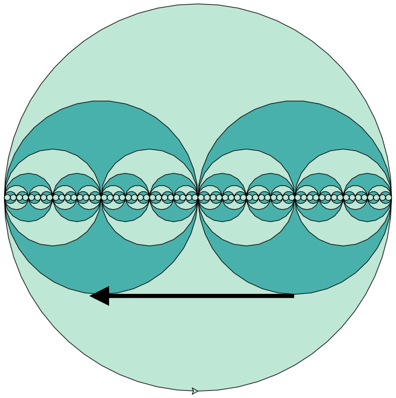
Nested Circles



- **Recursive case**
  - Draw the outer circle, add one to total
  - Position turtle for right recursive subcircle

```
def nestedCircles(radius, minRadius, colorOut, colorAlt):
    if radius < minRadius:
        return 0
    else:
        # contribute to the solution
        drawDisc(radius, colorOut)

        # save half of radius
        halfRadius = radius/2

        # position the turtle to draw right subcircle
        lt(90); fd(halfRadius); rt(90); fd(halfRadius)

        # draw right subcircle recursively
        right = nestedCircles(halfRadius, minRadius, colorAlt, colorOut)
```

Nested Circles



- **Recursive case**
  - Move the turtle to draw left subcircle recursively
  - (continued from previous slide)

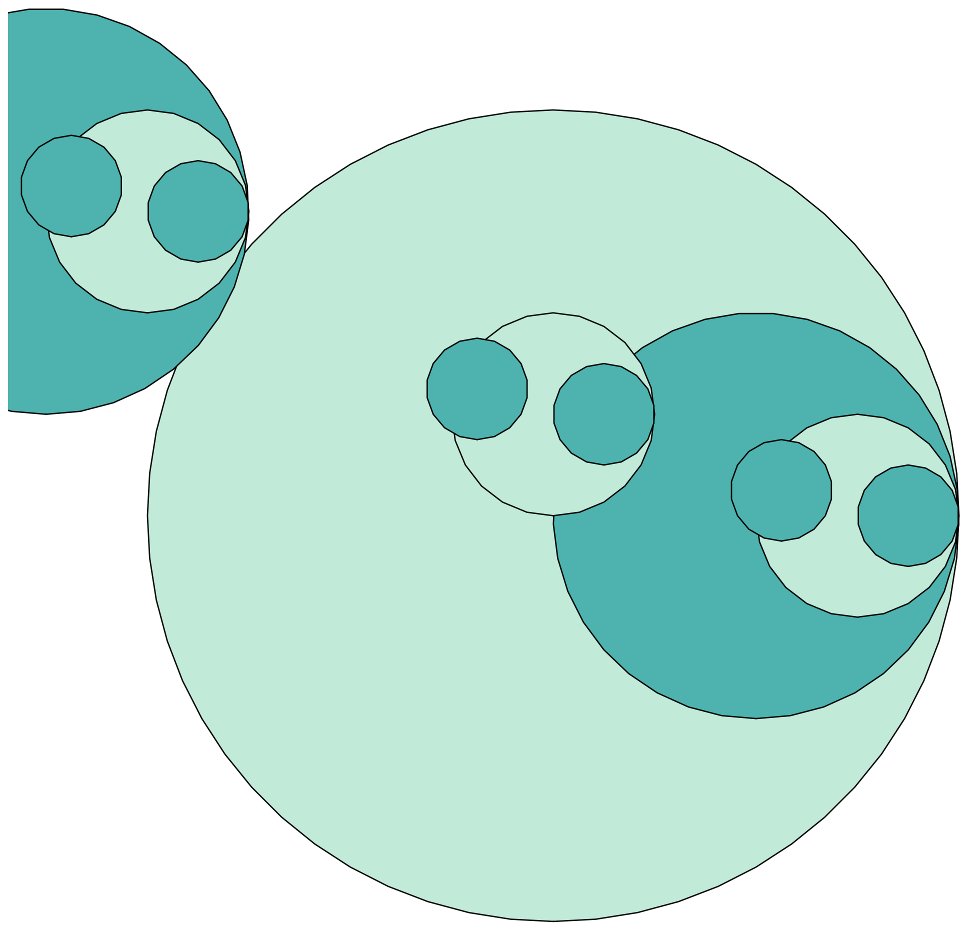
```
# draw right subcircle recursively
right = nestedCircles(halfRadius, minRadius, colorAlt, colorOut)

# position turtle for left subcircle
bk(radius)

# draw left subcircle recursively
left = nestedCircles(halfRadius, minRadius, colorAlt, colorOut)
```

Nested Circles

- **Recursive case**
  - Are we done? Let's try it!

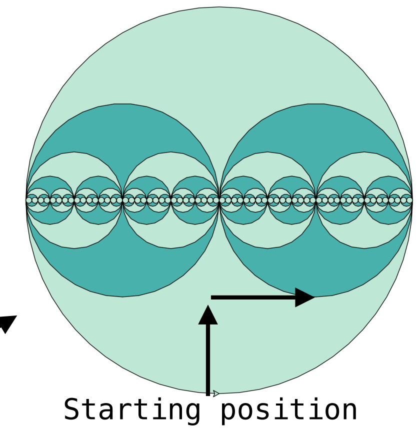
Nested Circles

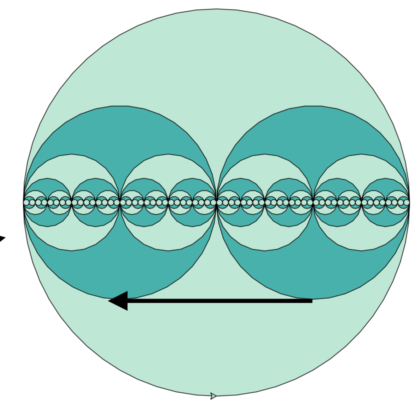
- **Recursive case**
  - Invariance matters! We **must** return the turtle to its starting state to make sure subsequent recursive calls behave correctly

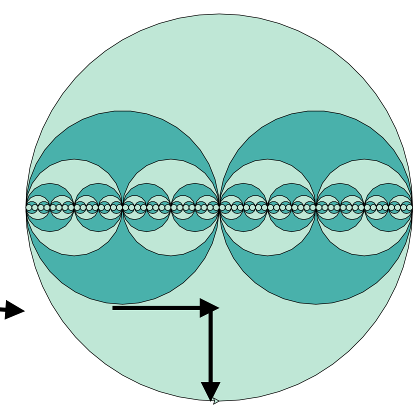


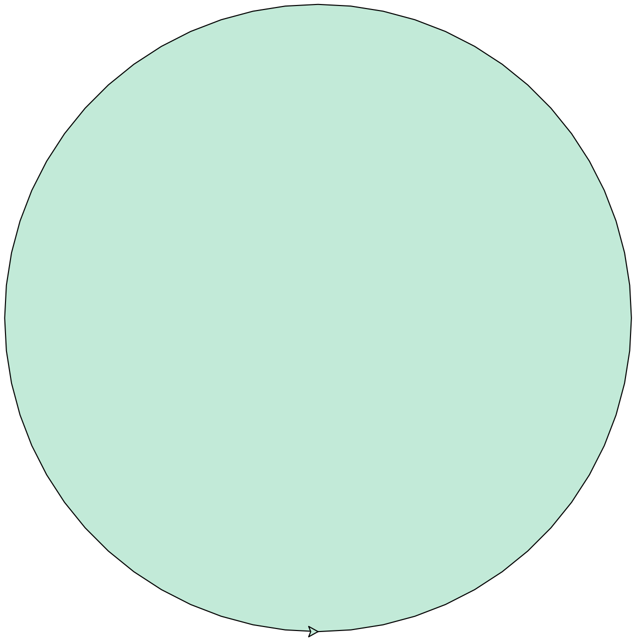
Maintaining Invariance

```python
def nestedCircles(radius, minRadius, colorOut, colorAlt):
    if radius < minRadius:
        return 0
    else:
        # contribute to the solution
        drawDisc(radius, colorOut)

        # save half of radius
        halfRadius = radius/2

        # position the turtle to draw right subcircle
        lt(90); fd(halfRadius); rt(90); fd(halfRadius)

        # draw right subcircle recursively
        right = nestedCircles(halfRadius, minRadius, colorAlt, colorOut)

        # position turtle for left subcircle
        bk(radius)

        # draw left subcircle recursively
        left = nestedCircles(halfRadius, minRadius, colorAlt, colorOut)

        # bring turtle back to start position
        fd(halfRadius);  lt(90);  bk(halfRadius);  rt(90)

        # return total number of circles drawn
        return 1 + right + left
```

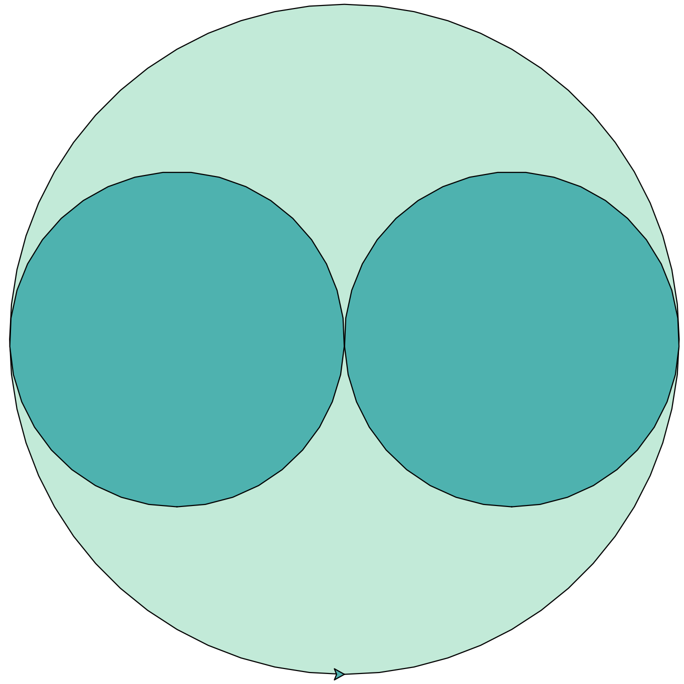




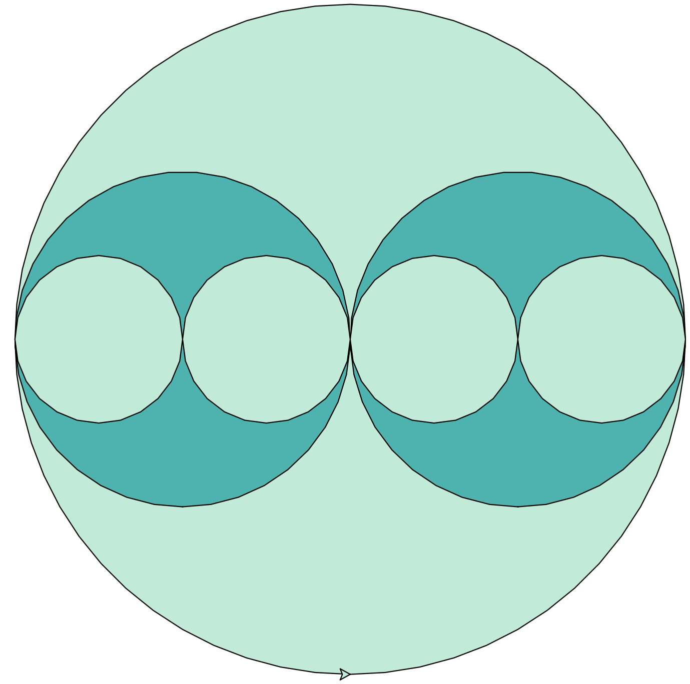




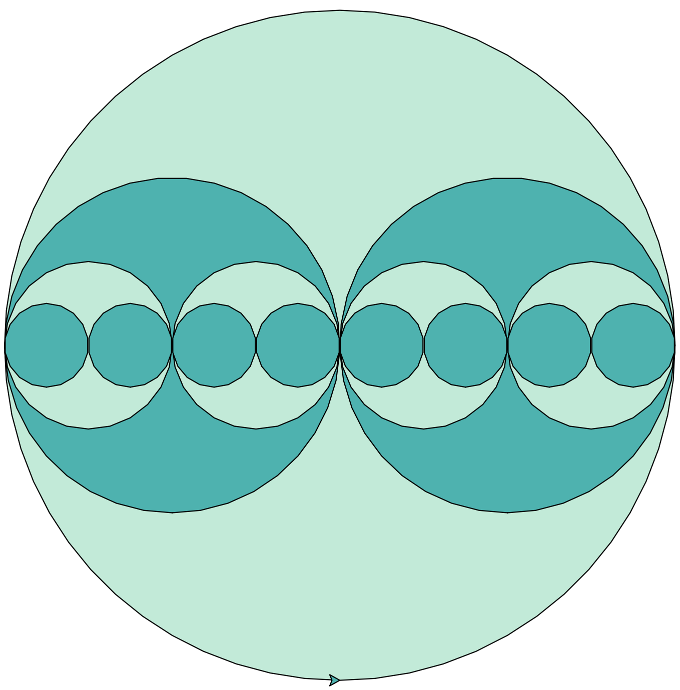
`nestedCircles(300, 300)`



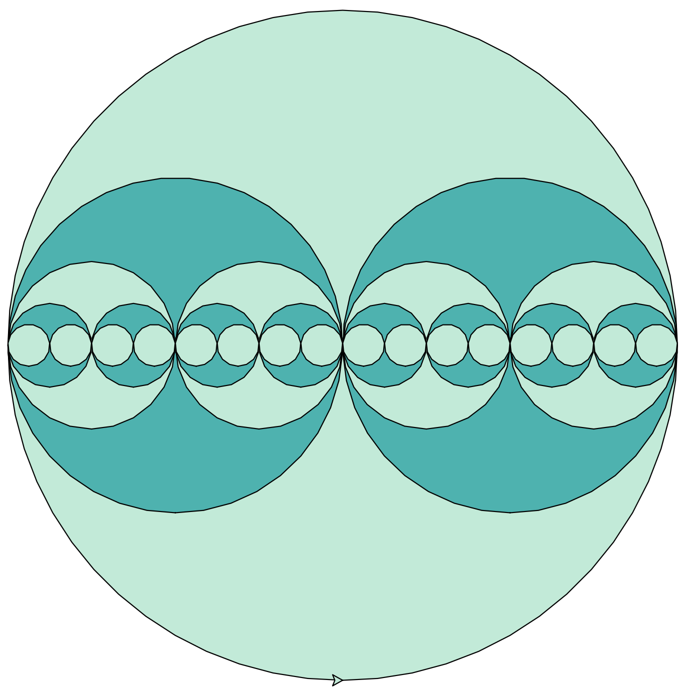
`nestedCircles(300, 150)`



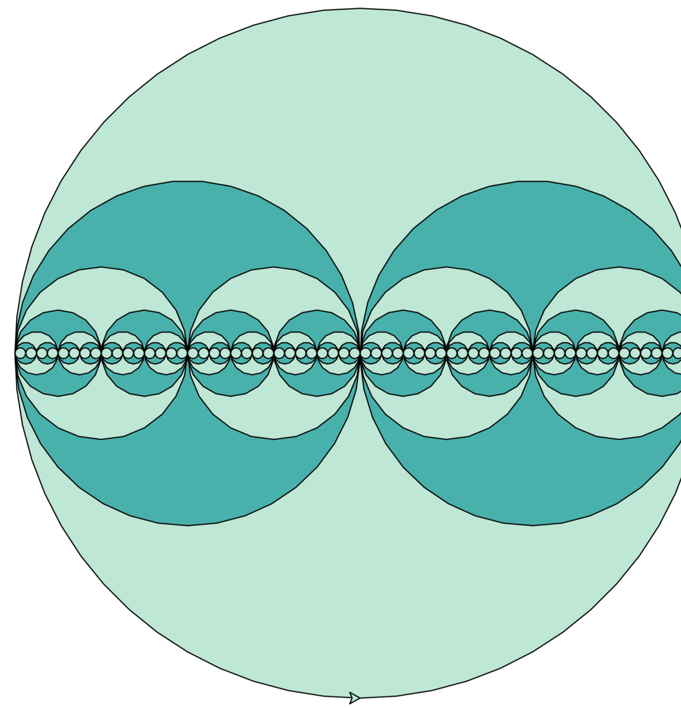
`nestedCircles(300, 75)`



`nestedCircles(300, 37)`



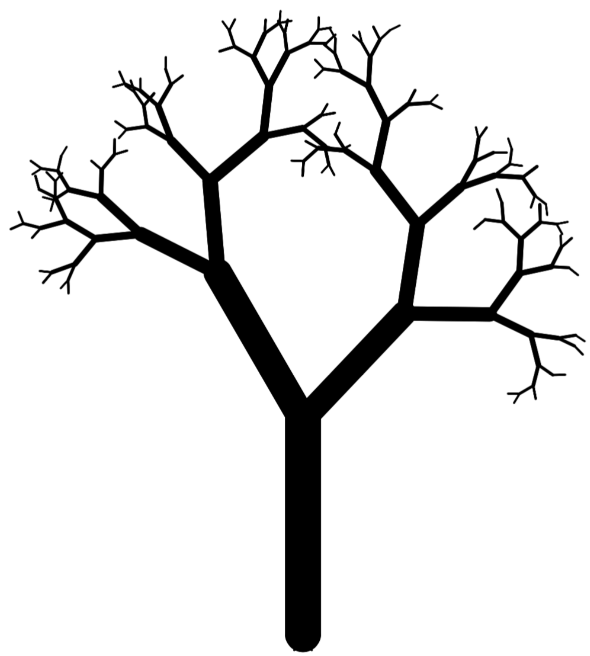
`nestedCircles(300, 9)`

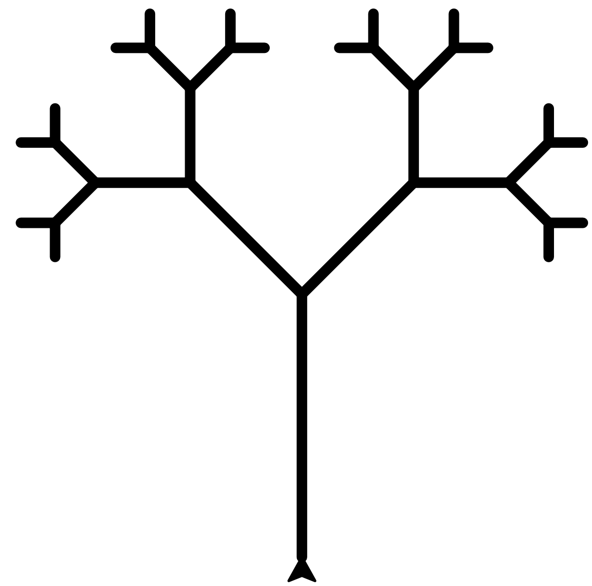


`nestedCircles(300, 2)`

Next Time

- Next time: We'll wrap up recursion with a few more examples and compare to iterative approaches





The end!