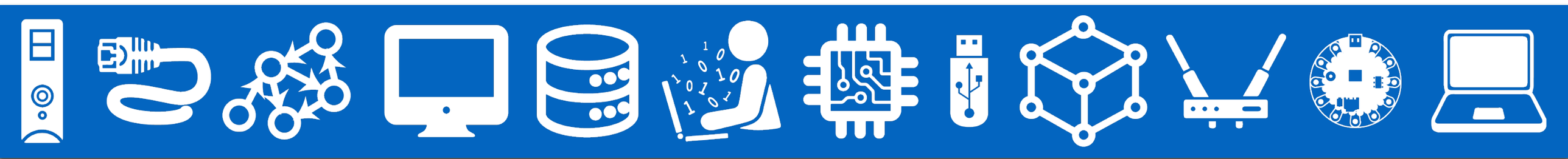
CS134:
Graphical Recursion



Announcements & Logistics

- Lab 7 (Recursion) will be released today
  - There is a Pre-Lab - **do this individually**, not with a partner!
  - We will collect these on paper! **Bring them on paper**!
- **HW 6** due Monday on Gradescope
- Lab 7, 8, and 9 are **partner labs**
  - Pair programming is an important skill as well as a vehicle for learning
  - **Partner? Must attend one lab session together**
- **Final Exam**: Wednesday, December 11 at 9:30am in Wachenheim B11

Do You Have Any Questions?

Pair Programming: Best Practices

- Goal is to work together as a team using **one computer**:
  - Driver: controls the keyboard (handles the details, debugging, etc)
  - Navigator: helps guide (big picture person, decides where to go)
- **Both roles are important** and you both should switch roles often!
- **Communication is key**: but be polite, respectful and patient
- Discussing high level strategies and goals early, before writing the code, helps avoid bugs as well as conflict
- Resources:
  - How Pair Programming Really Works by Stuart Wray
  - Longish article on pair programming pros, cons, and strategies

CS Colloquium Today

- **CS Pre-Registration Info Session**
- 2:35pm in Wege Auditorium
- Come learn about what's on offer for Winter Study + Spring Semester!

CS10 :: Unix and Software Tools

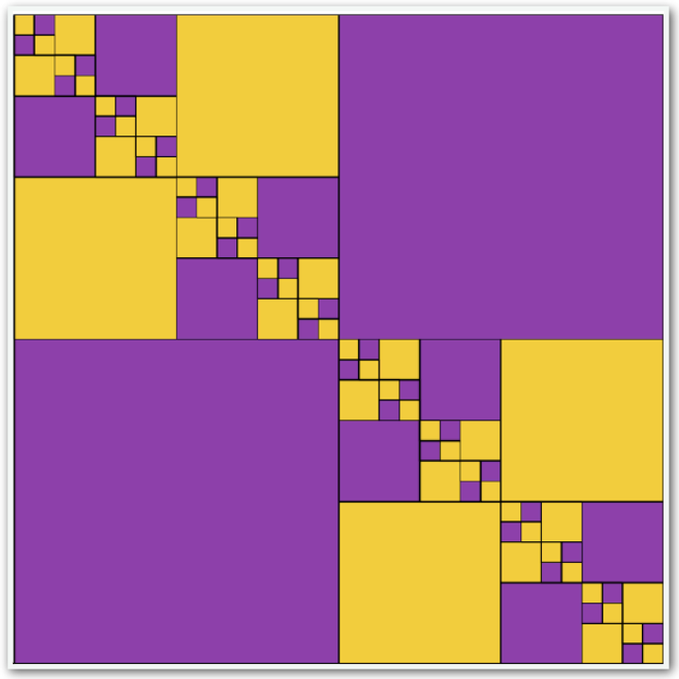
This course serves as a guided introduction to the Unix operating system and a variety of software tools. Students in this course will work on Unix workstations, available in the Department's laboratory. By the end of the course, students will be familiar with Unix and will be able to use Git as a collaborative tool. As a final project, students will work together in teams to explore an API of their choice. The exact topics to be covered may vary depending upon the needs and desires of the students. The course is designed for individuals who understand basic program development techniques as discussed in an introductory programming course (Computer Science 134 or equivalent), but who wish to become familiar with a broader variety of computer systems and programming languages. This course is not intended for students who have completed a course at the 200 level or above.

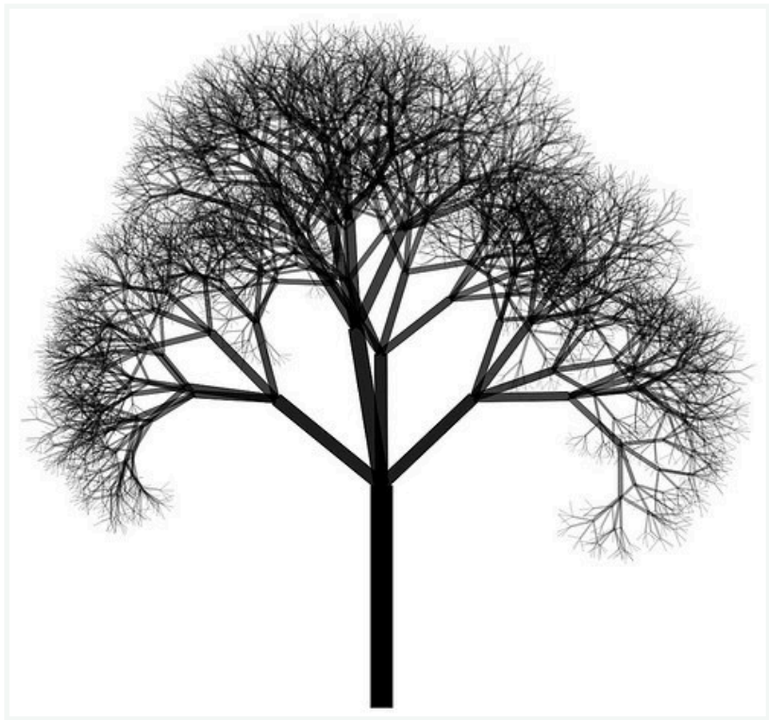
CS16 :: Intro to the CS Research Process

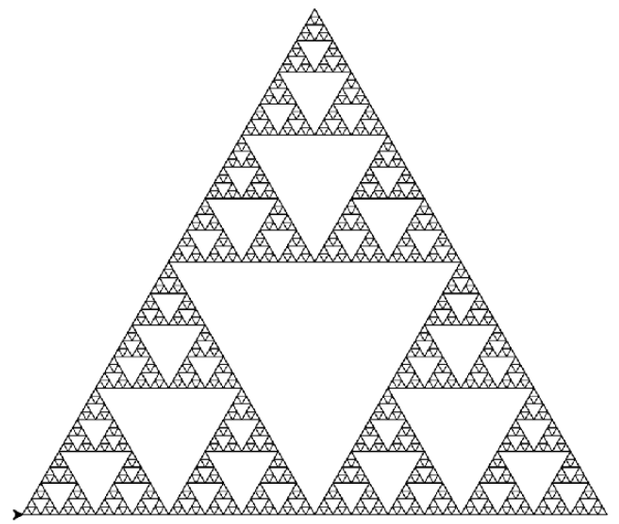
This course introduces students to the research process in Computer Science. Students will learn how to critically read research papers and to find relevant related work. They will also learn about experimental design and data visualization. Students will apply these skills in the context of a specific research paper, recreating some of the data collection, analysis, and data visualization from that paper. A flipped classroom approach will be used, with students watching recorded videos outside of class in preparation for in-class discussions and activities. Students will create a written research project proposal that describes how they plan to extend the research paper to answer a different question, including describing how the existing experimental framework would need to be modified and what experiments would need to be conducted. Assessment will be based on this written project proposal and an in-class oral presentation of that proposal.

Last Time

- Continuing recursion
  - Alternative to iteration
  - New problem solving paradigm
- Function frame model to understand recursion behind the scenes





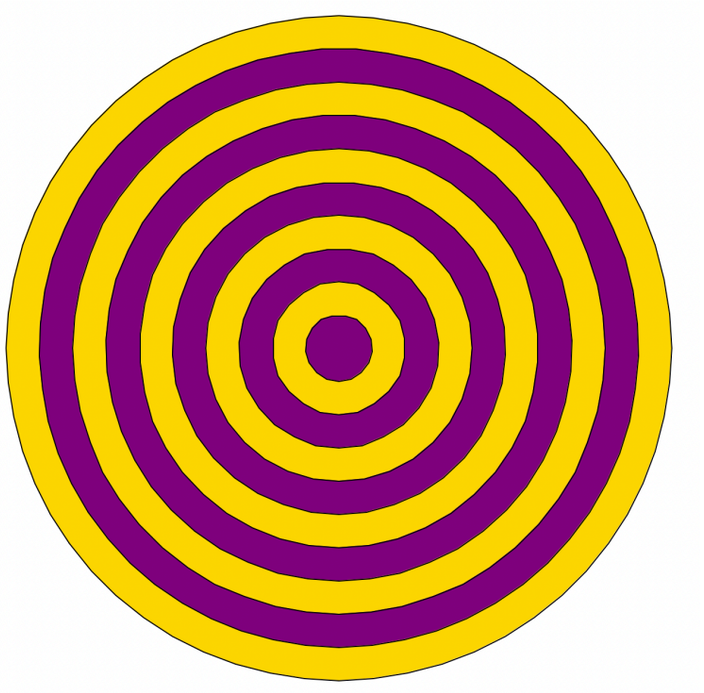


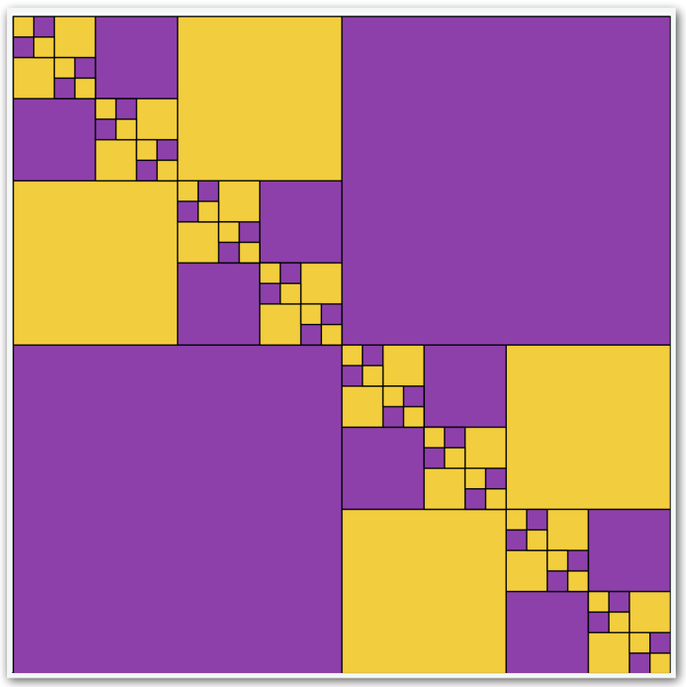
Last Time: Recursive Approach to Problem Solving

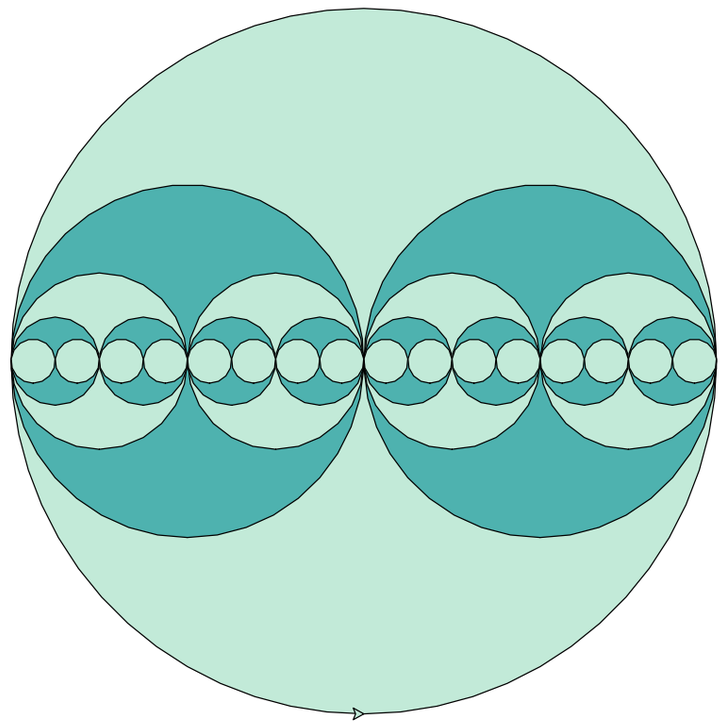
- A recursive function is a function **that calls itself**
- A recursive approach to problem solving has two main parts:
  - **Base case(s).** When the problem is **so small**, we solve it directly, without having to reduce it any further
  - **Recursive step.** Does the following things:
    - Performs 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.)

```
pen down
```

Turtle In Python

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

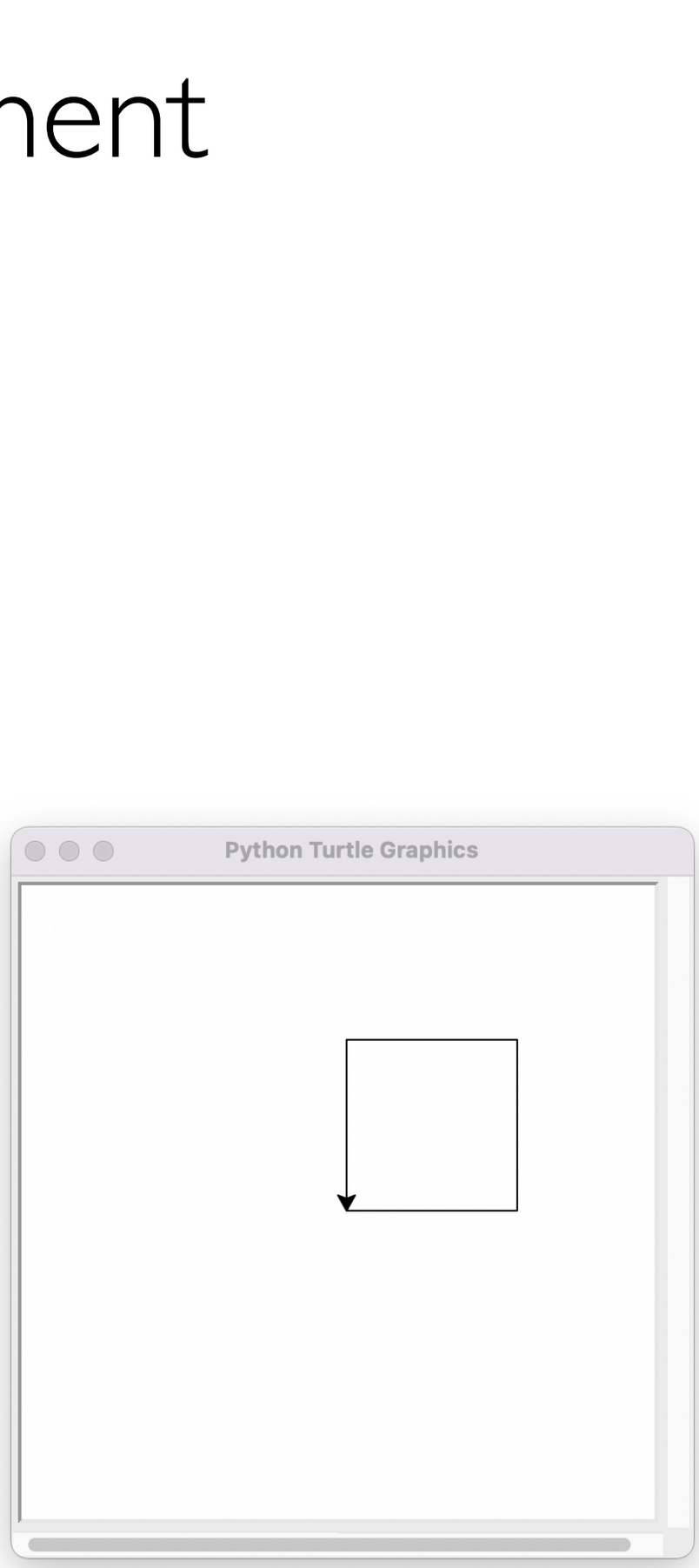
Use `from turtle import *` to use these commands

`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 lowers pen from belly
`shape(shp)`	sets the turtle's shape to shp
`speed(spd)`	sets the turtle's speed 1-10 (slow-fast). 0 skips animation.
`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()
```

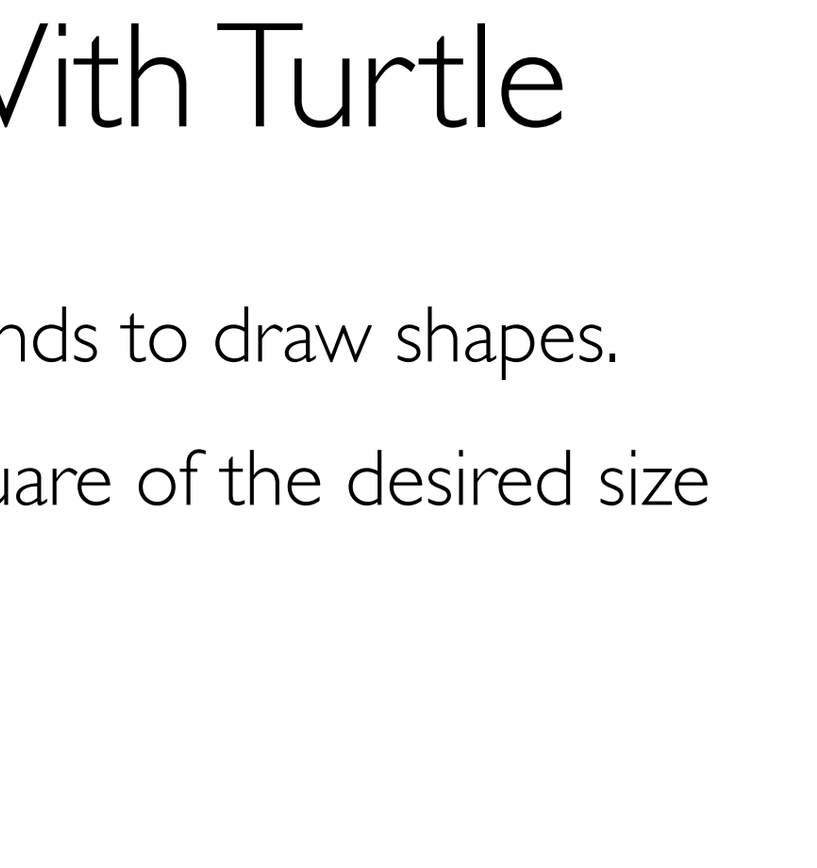


Drawing Basic Shapes With Turtle

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

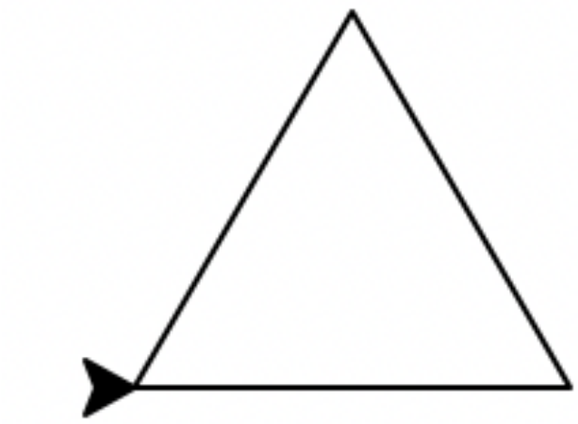
```
setup(400, 400)
reset()
draw_square(150)
```



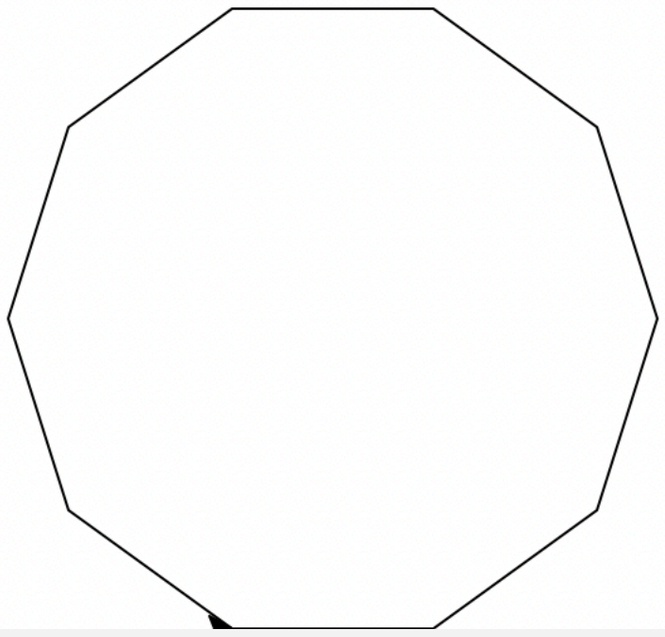
Drawing Basic Shapes With Turtle

- How about drawing polygons?

```
def draw_polygon(length, num_sides):
    for i in range(num_sides):
        fd(length)
        lt(360/num_sides)
    done()
```



`draw_polygon(80, 3)`

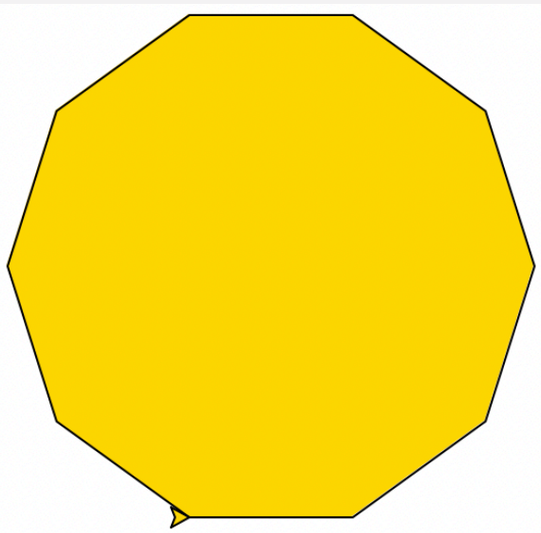


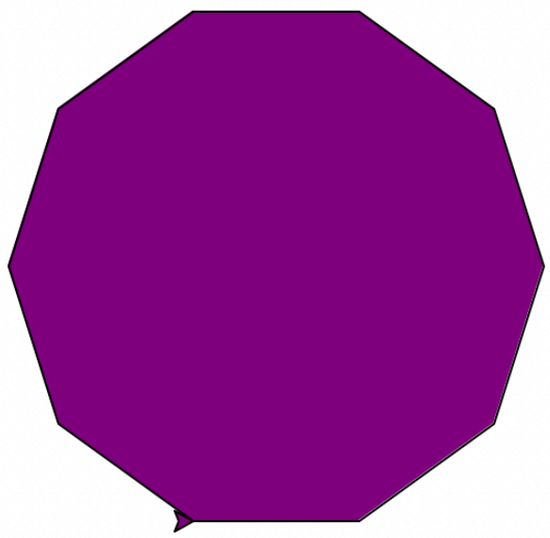
`draw_polygon(80, 10)`

Adding Color!

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

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

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



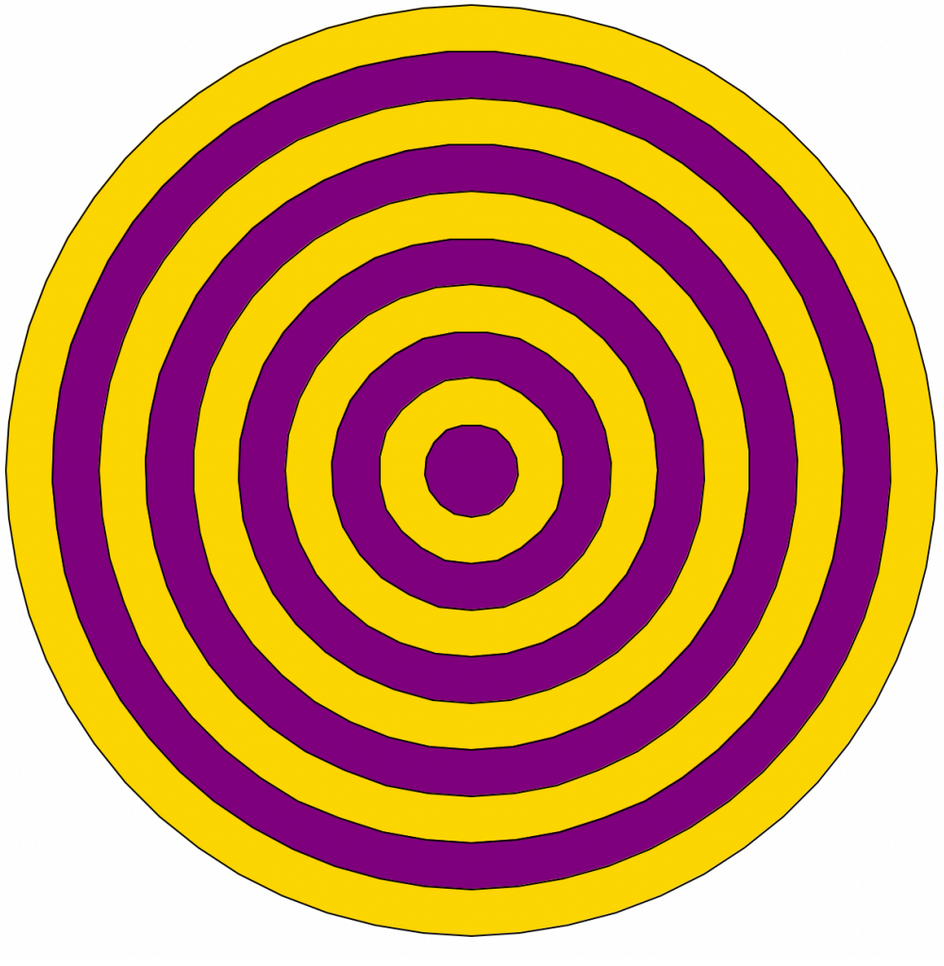


```
draw_polygon_color(80, 10, "gold")
```

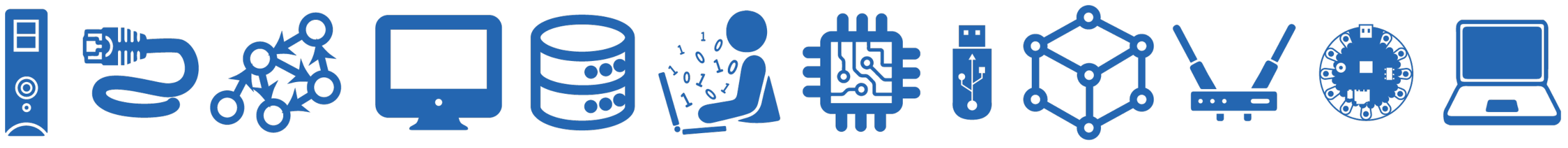
```
draw_polygon_color(80, 10, "purple")
```

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)
  - Let's also count the number of circles we draw, so add one to our count!

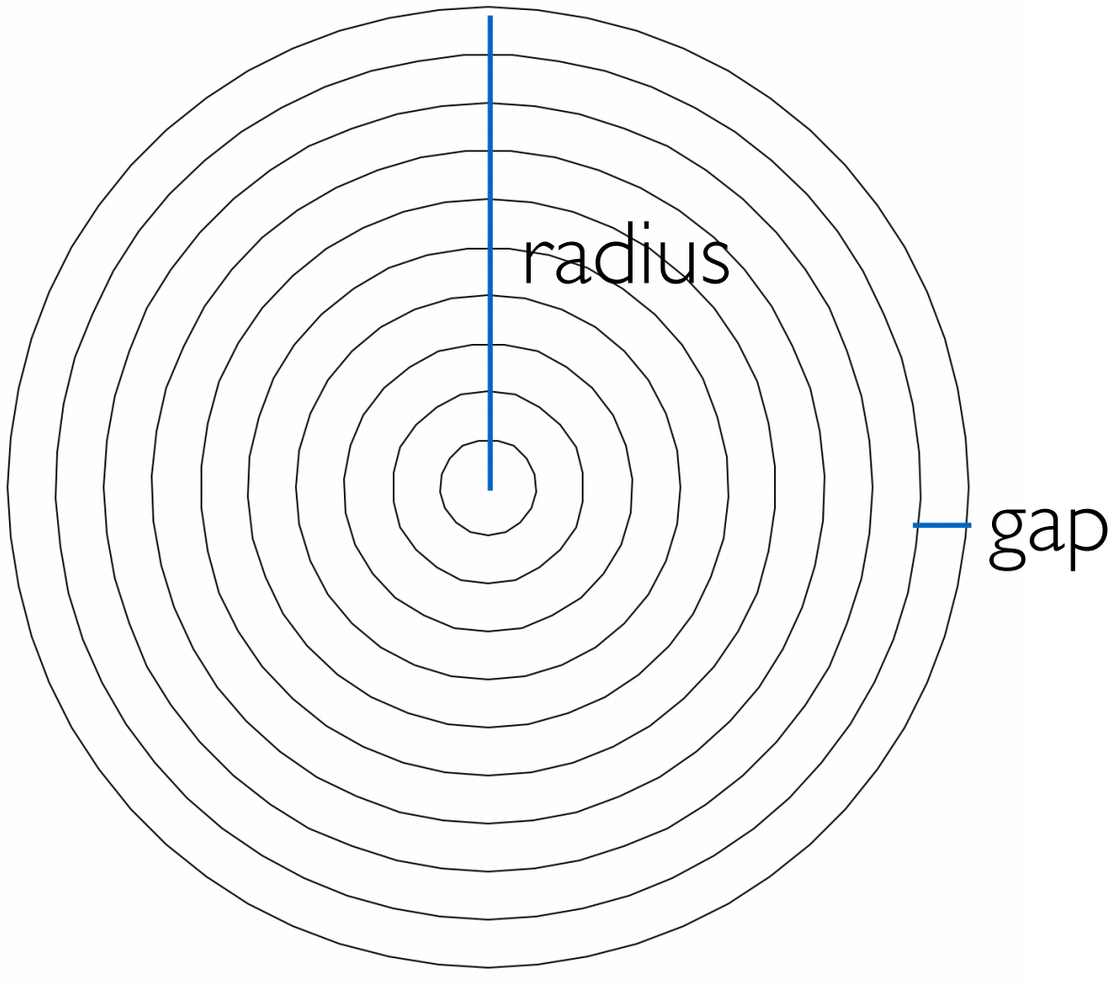
Counting the number of circles isn't necessary for drawing pictures, but it does make debugging easier!

Concentric Circles

- Function definition

```
concentric_circles(radius, gap)
```

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



Concentric Circles

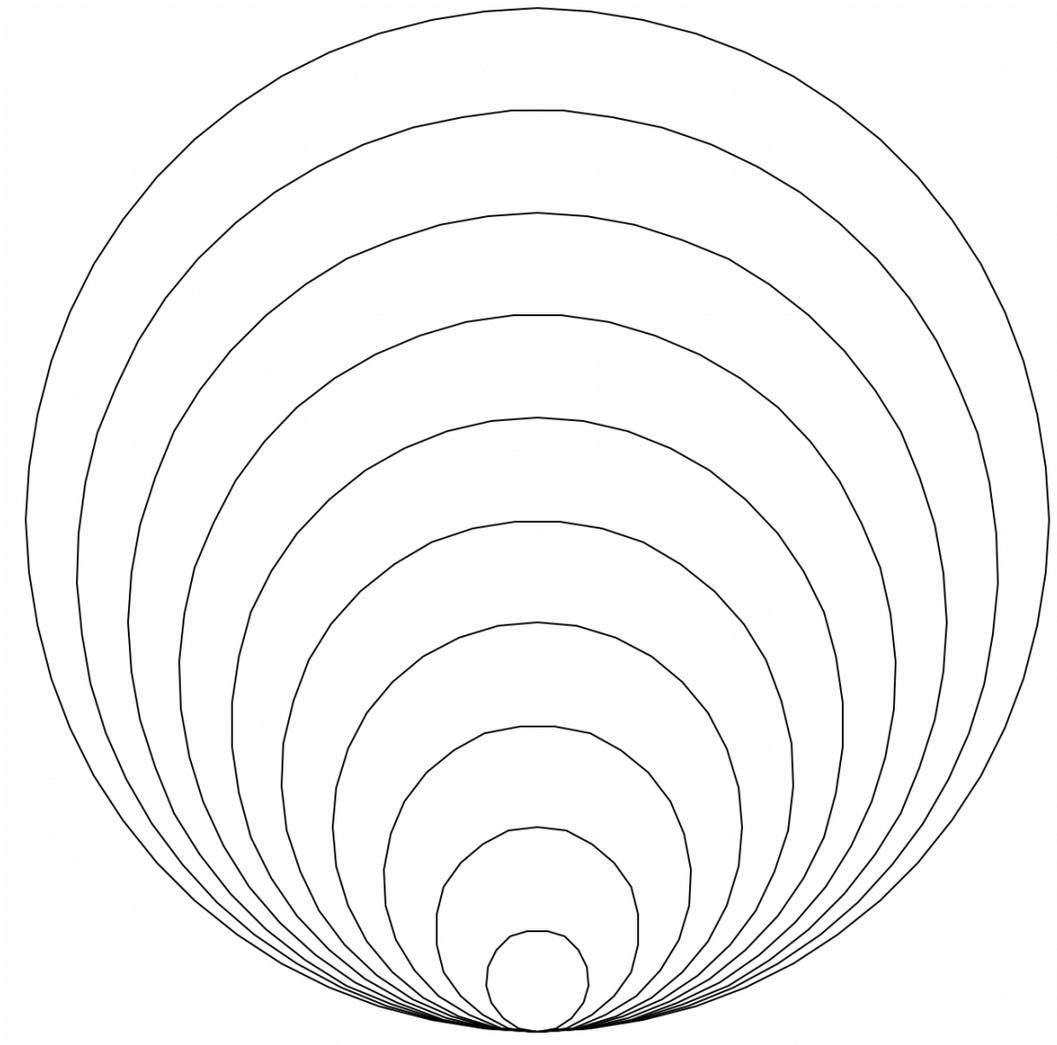
```
def concentric_circles(radius, gap):
    # base case, don't draw anything, return 0
    if radius < gap:
        return 0
    else:
        # tell the turtle draw a circle
        circle(radius)

        # recursive function call; draw smaller circles
        num = concentric_circles(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

```
print("Num Circles:", concentric_circles(300, 30))
```

```
Num Circles: 10
```

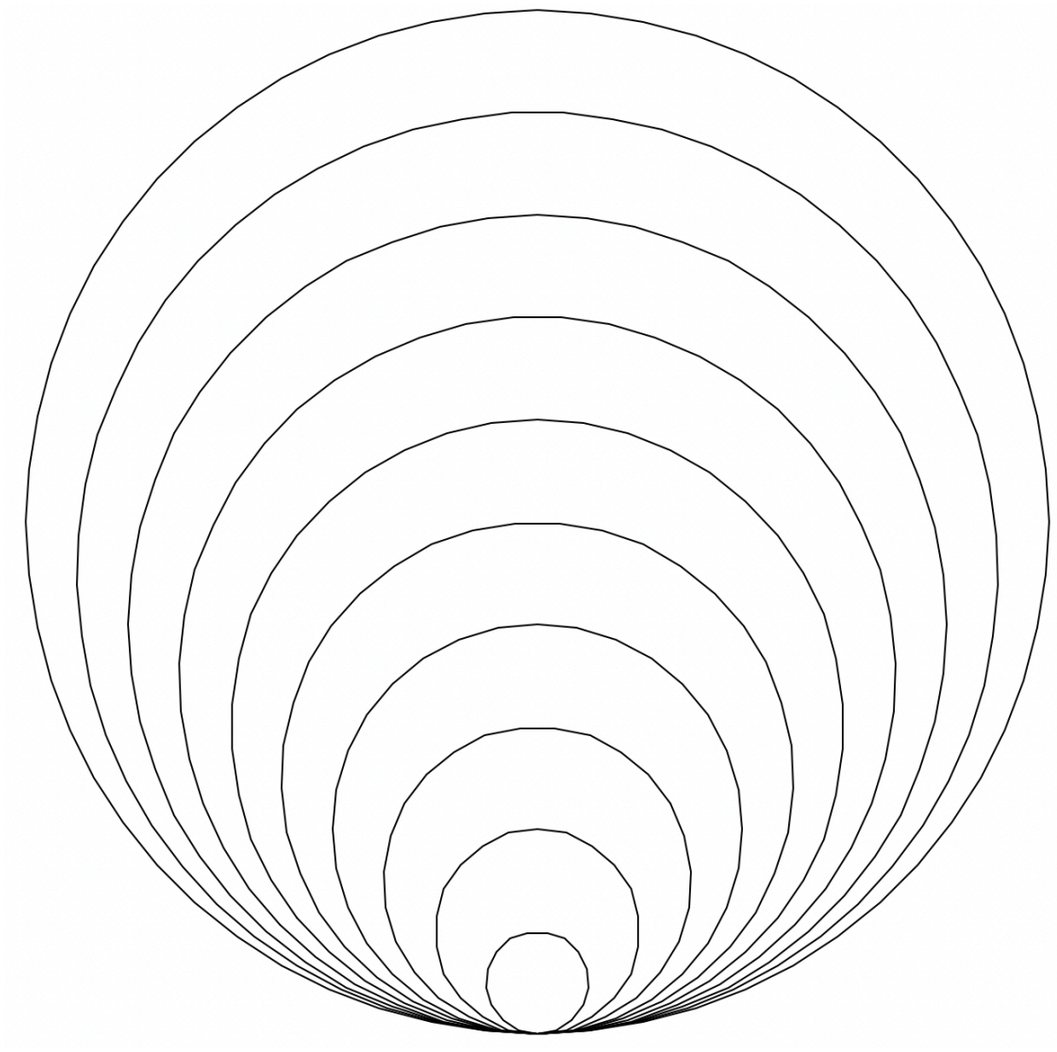


- Pretty picture, and almost there! But not quite right. What happened?

Concentric Circles

```
print("Num Circles:", concentric_circles(300, 30))
```

```
Num Circles: 10
```

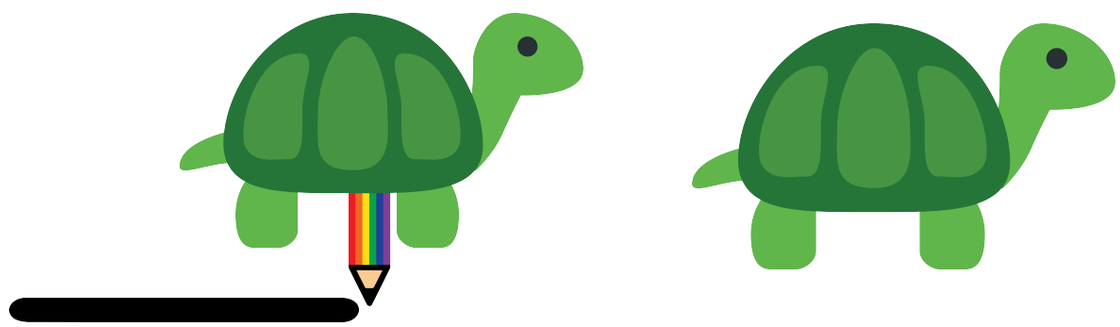


- We need to reposition the turtle after each recursive call.

Concentric Circles

```
def concentric_circles(radius, gap):
    # base case, don't draw anything
    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 = concentric_circles(radius-gap, gap)

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

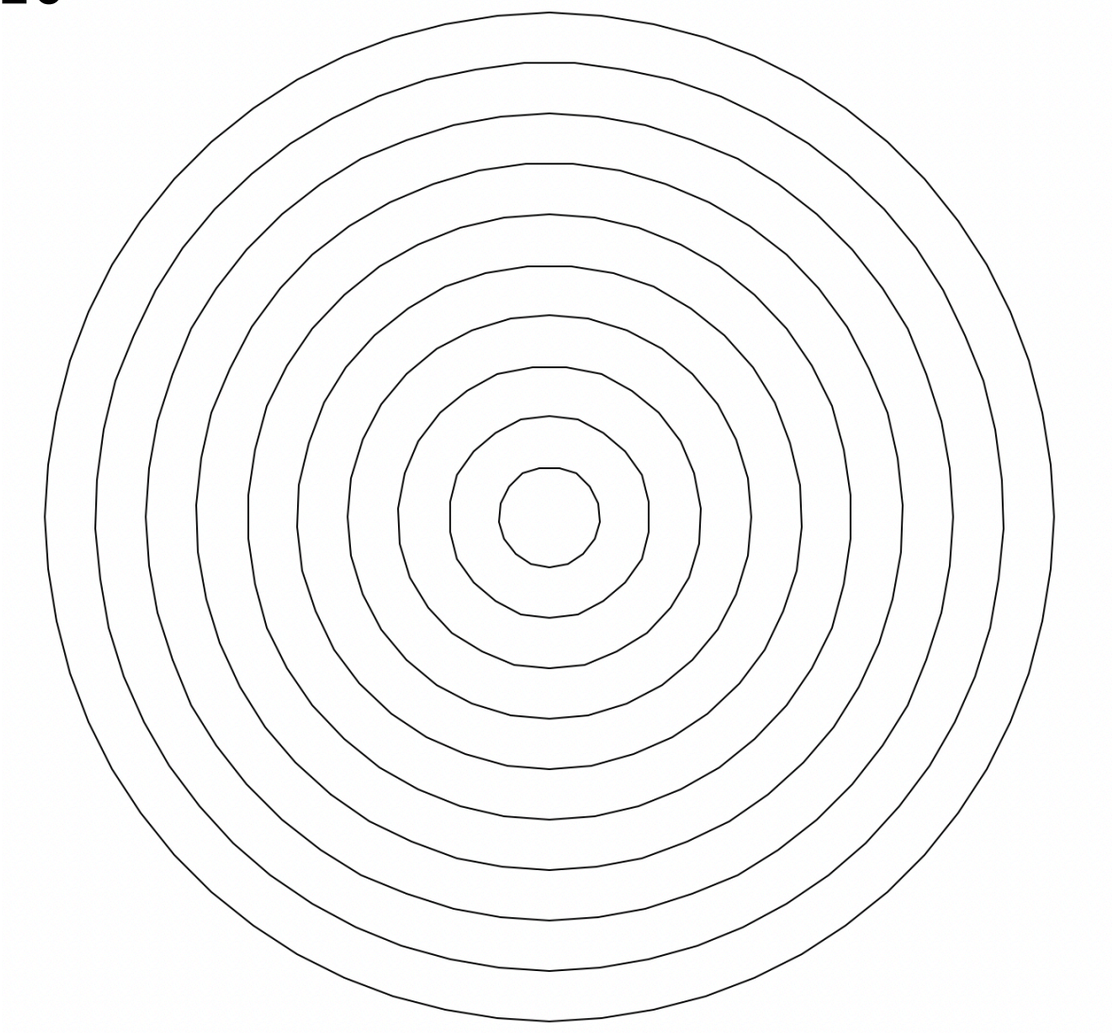
Turtle: pen commands



Concentric Circles

```
print("Num Circles:", concentric_circles(300, 30))
```

```
Num Circles: 10
```



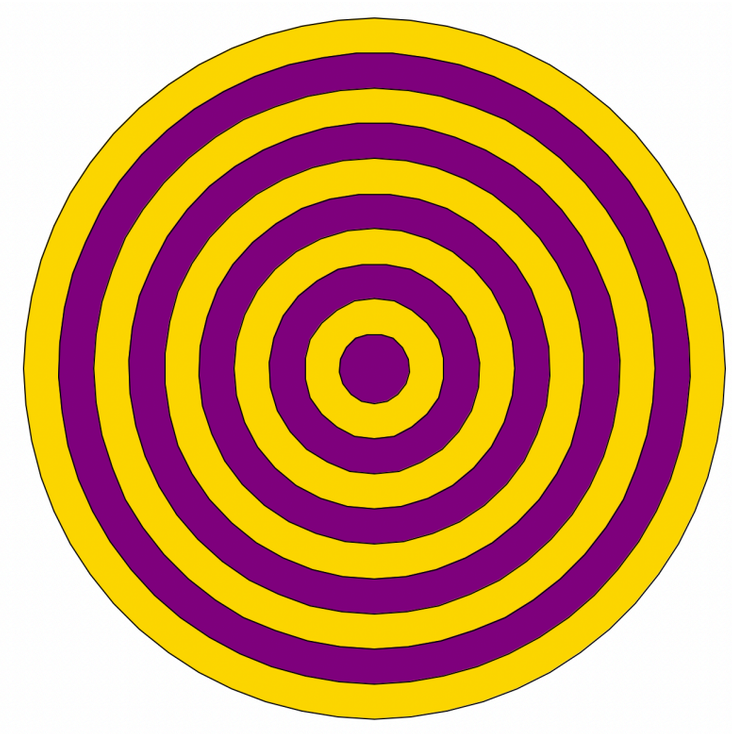
- Great! Now let's add some color.

Concentric Circles With Colors

- Function definition

```
concentric_circles(radius, gap, color_outer, color_inner)
```

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



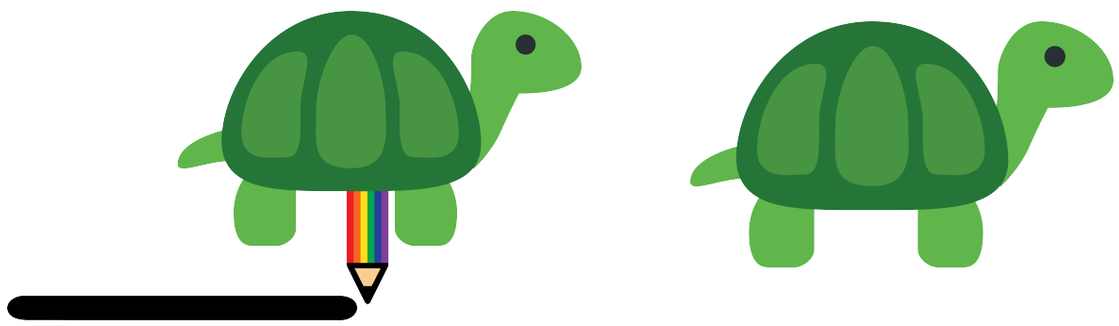
Concentric Circles: Adding Color

- Base case and recursive case stay the same
- How do we achieve the alternating colors?
- Just swap the order in the recursive call
  - `color_outer` becomes `color_inner` 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

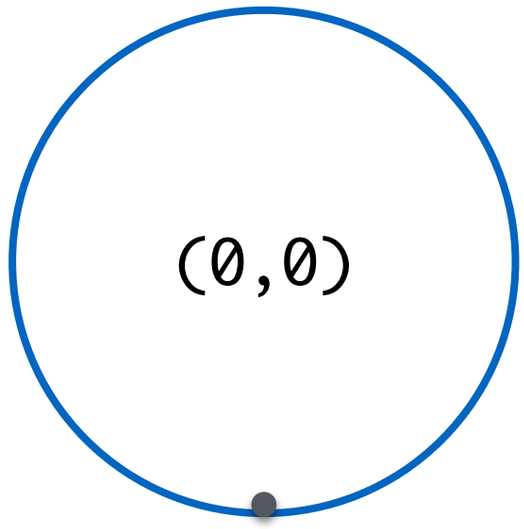
Helper Function

```
def draw_disc(radius, color):
    """
    Draw circle of a given radius
    and fill it with 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: pen commands



`down()` `up()`

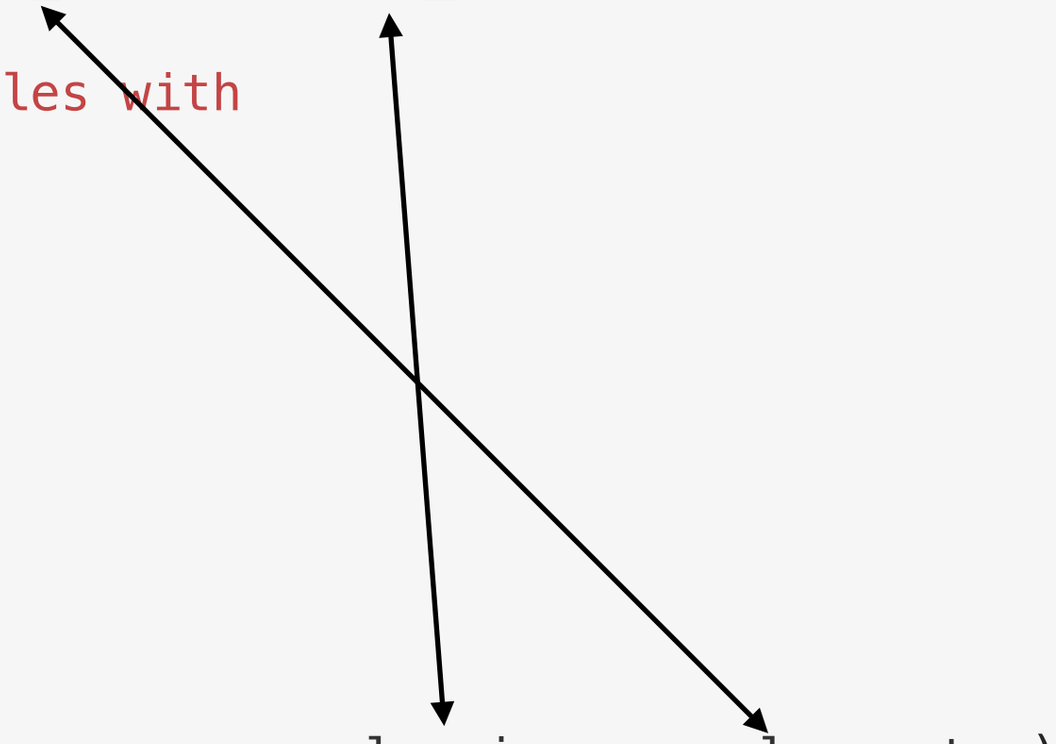


Starting position of turtle

`(0, -radius)`

The Recursive Function

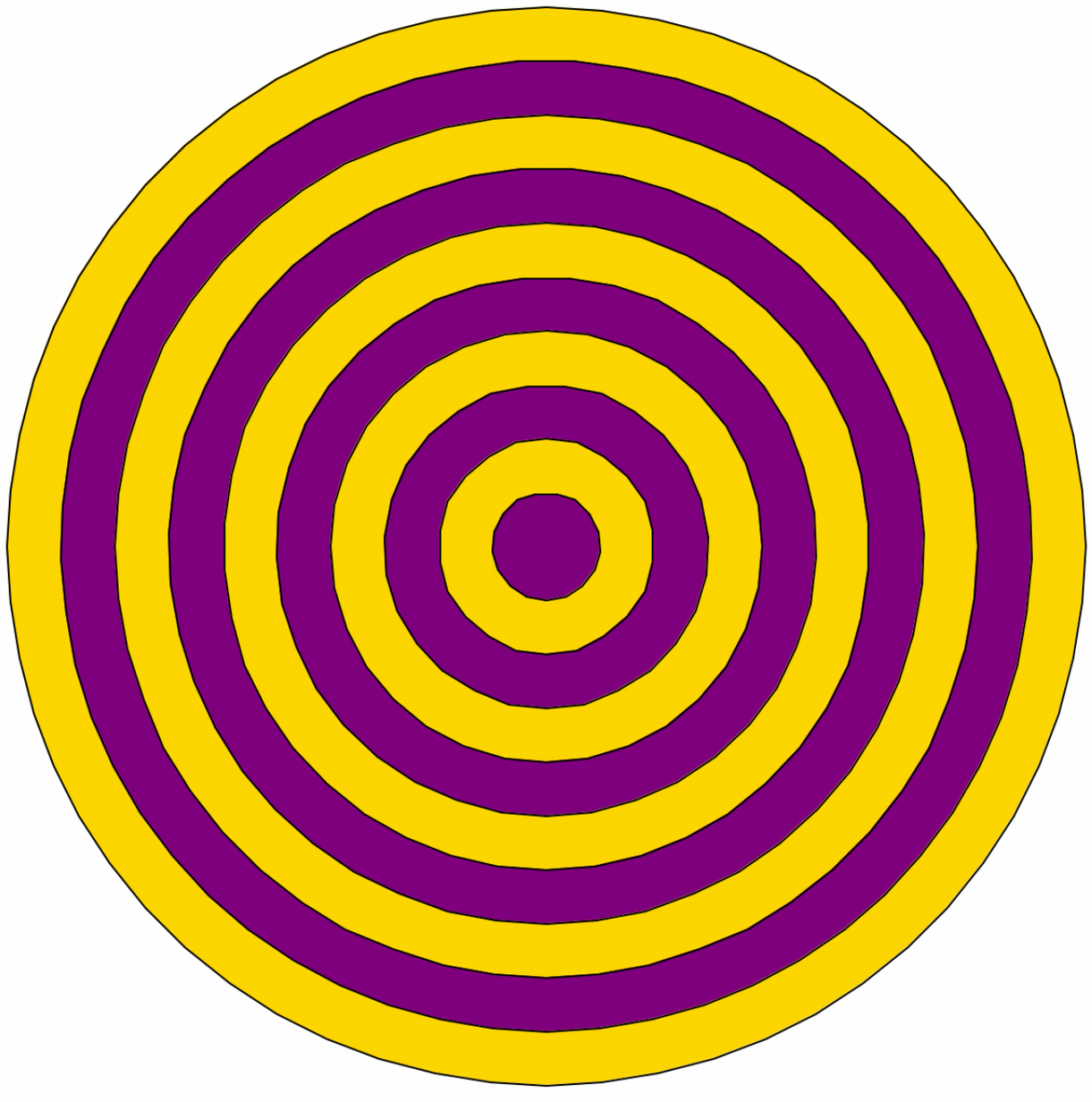
```
def concentric_circles_color(radius, gap, color_outer, color_inner):
    """
    Recursive function to draw concentric circles with
    alternating colors
    """
    # base case, don't draw anything, return 0
    if radius < gap:
        return 0
    else:
        draw_disc(radius, color_outer)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



Concentric Circles

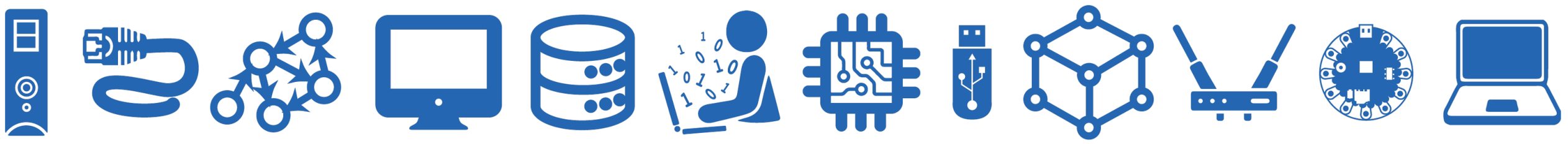
```
print("Num circles:", concentric_circles_color(300, 30, "gold", "purple"))
```

```
Num Circles: 10
```

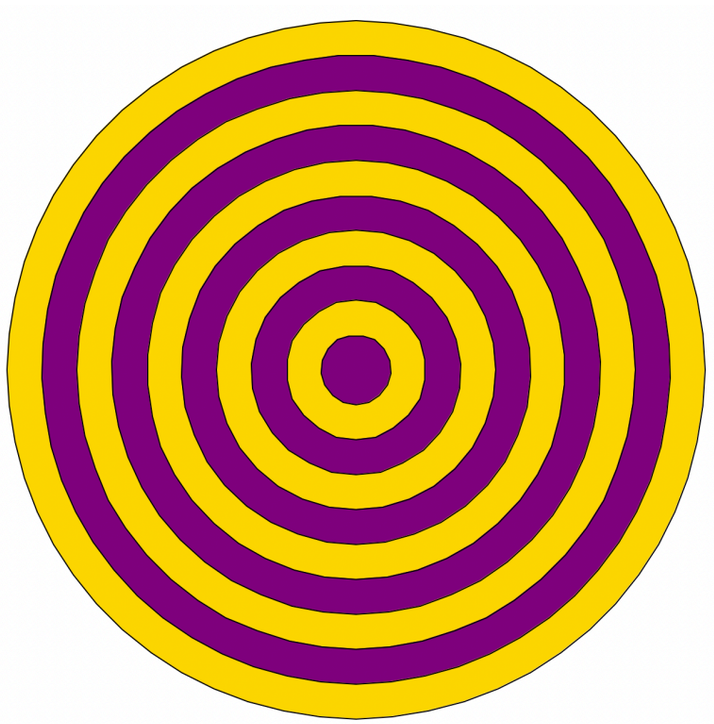


Function Frame Model:

`concentric_circles`



```
def concentric_circles(radius, gap, color_outer, color_inner):
  """Recursive function to draw concentric circles"""
  if radius < gap:
   return 0
  else:
   draw_disc(radius, color_out)
    lt(90); fd(gap); rt(90)
   num = concentric_circles(radius-gap, gap, color_in, color_out)
   return 1 + num
```



```
def concentric_circles(radius, gap, color_out, color_in):
  """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    draw_disc(radius, color_out)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(radius-gap, gap, color_in, color_out)
    return 1 + num
```

```
>>> concentric_circles(18, 5, "purple", "gold")
```

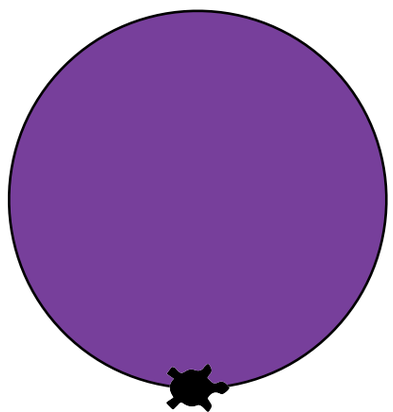
```
def concentric_circles(radius, gap, color_out, color_in):
  """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    draw_disc(radius, color_out)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(radius-gap, gap, color_in, color_out)
    return 1 + num
```

```
>>> concentric_circles(18, 5, "purple", "gold")
```

contrc_circles(18,5,'p','g')

radius 18 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```



```
def concentric_circles(radius, gap, color_out, color_in):
  """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    draw_disc(radius, color_out)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(radius-gap, gap, color_in, color_out)
    return 1 + num
```

```
>>> concentric_circles(18, 5, "purple", "gold")
```

contrc_circles(18,5,'p','g')

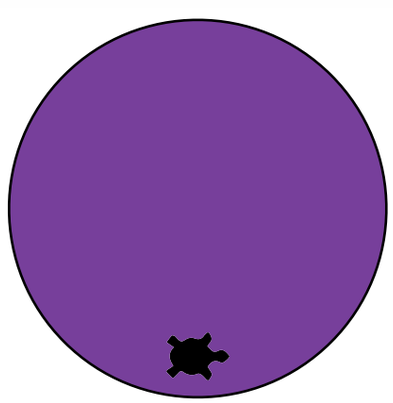
radius 18 gap 5

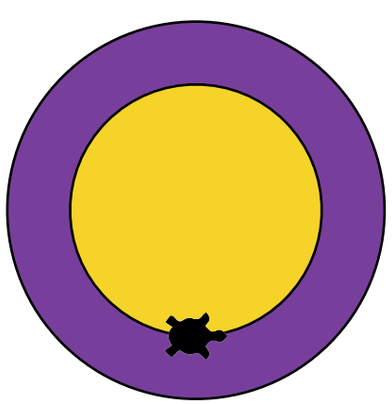
```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(13,5,'g','p')

radius 13 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```





```
def concentric_circles(radius, gap, color_out, color_in):
  """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    draw_disc(radius, color_out)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(radius-gap, gap, color_in, color_out)
    return 1 + num
```

```
>>> concentric_circles(18, 5, "purple", "gold")
```

contrc_circles(18,5,'p','g')

radius 18 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(13,5,'g','p')

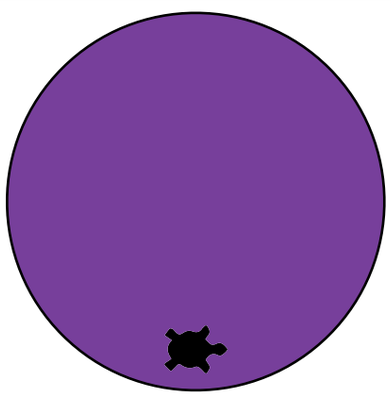
radius 13 gap 5

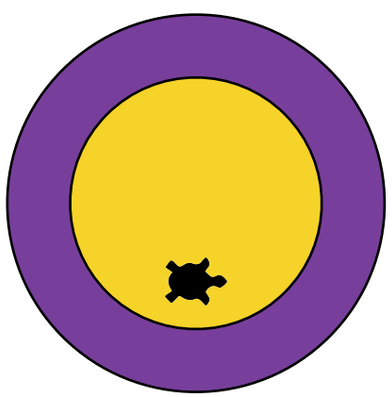
```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```

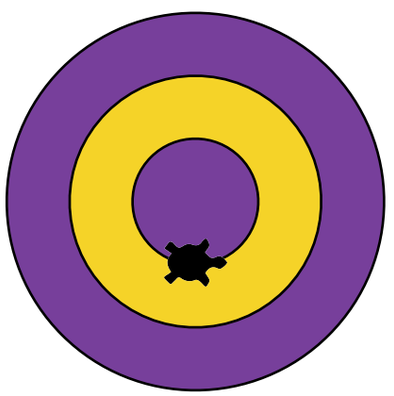
contrc_circles(8,5,'p','g')

radius 8 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```







```
def concentric_circles(radius, gap, color_out, color_in):
  """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    draw_disc(radius, color_out)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(radius-gap, gap, color_in, color_out)
    return 1 + num
```

```
>>> concentric_circles(18, 5, "purple", "gold")
```

contrc_circles(3,5,'g','p')

radius 3 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(18,5,'p','g')

radius 18 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(13,5,'g','p')

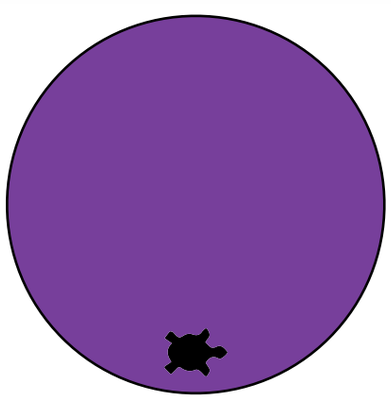
radius 13 gap 5

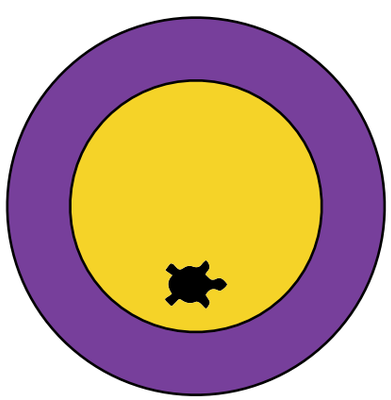
```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```

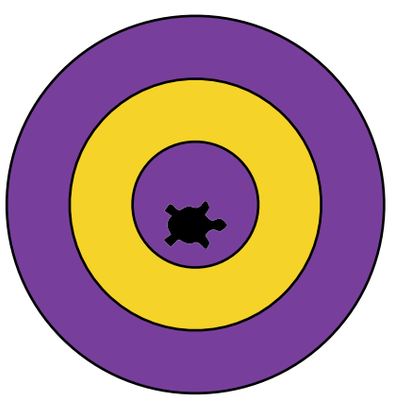
contrc_circles(8,5,'p','g')

radius 8 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```







```
def concentric_circles(radius, gap, color_out, color_in):
  """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    draw_disc(radius, color_out)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(radius-gap, gap, color_in, color_out)
    return 1 + num
```

```
>>> concentric_circles(18, 5, "purple", "gold")
```

contrc_circles(18,5,'p','g')

radius 18 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(13,5,'g','p')

radius 13 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(8,5,'p','g')

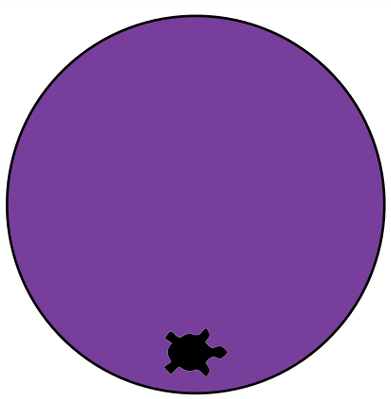
radius 8 gap 5

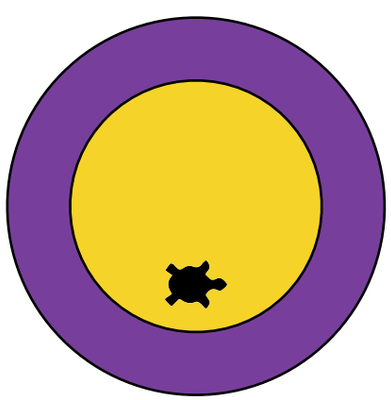
```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (0, g, clr_i, clr_o)
  return 1 + num
```

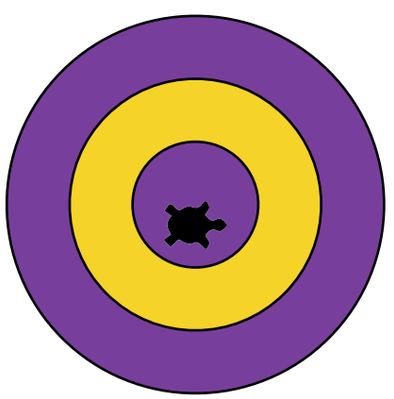
contrc_circles(3,5,'g','p')

radius 3 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```







```
def concentric_circles(radius, gap, color_out, color_in):
  """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    draw_disc(radius, color_out)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(radius-gap, gap, color_in, color_out)
    return 1 + num
```

```
>>> concentric_circles(18, 5, "purple", "gold")
```

contrc_circles(18,5,'p','g')

radius 18 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(13,5,'g','p')

radius 13 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(8,5,'p','g')

radius 8 gap 5

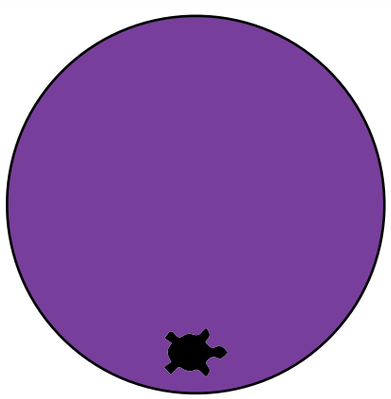
```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```

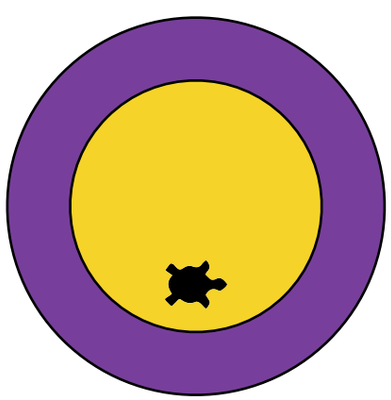
0

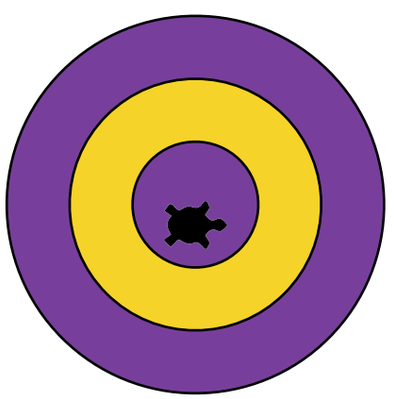
contrc_circles(3,5,'g','p')

radius 3 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```







```
def concentric_circles(radius, gap, color_out, color_in):
  """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    draw_disc(radius, color_out)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(radius-gap, gap, color_in, color_out)
    return 1 + num
```

```
>>> concentric_circles(18, 5, "purple", "gold")
```

contrc_circles(18,5,'p','g')

radius 18 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(13,5,'g','p')

radius 13 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = 1 entric_circles
        -g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(8,5,'p','g')

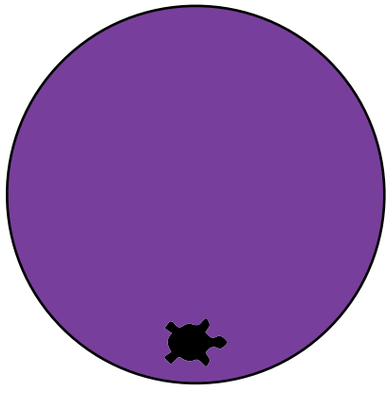
radius 8 gap 5

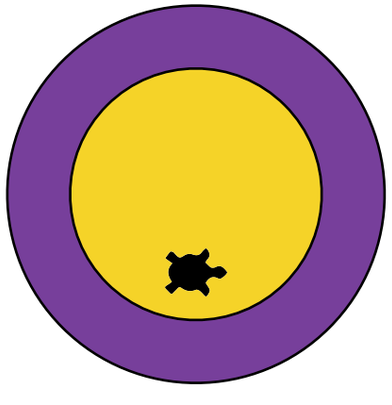
```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); ap); rt(90)
  num = c 0 tric_circles
        (  , g, clr_i, clr_o)
  return 1 + num
```

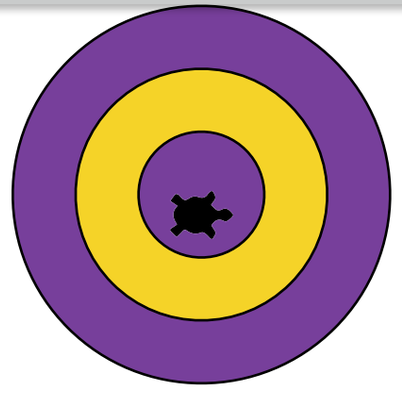
contrc_circles(3,5,'g','p')

radius 3 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```







```
def concentric_circles(radius, gap, color_out, color_in):
  """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    draw_disc(radius, color_out)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(radius-gap, gap, color_in, color_out)
    return 1 + num
```

```
>>> concentric_circles(18, 5, "purple", "gold")
```

contrc_circles(18,5,'p','g')

radius 18 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(13,5,'g','p')

radius 13 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(8,5,'p','g')

radius 8 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```

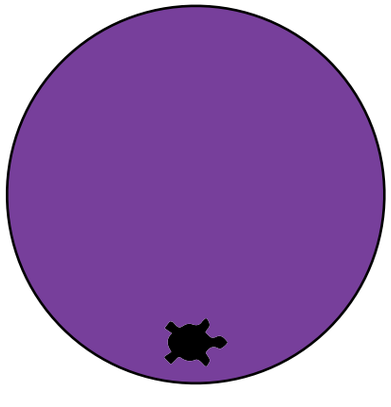
contrc_circles(3,5,'g','p')

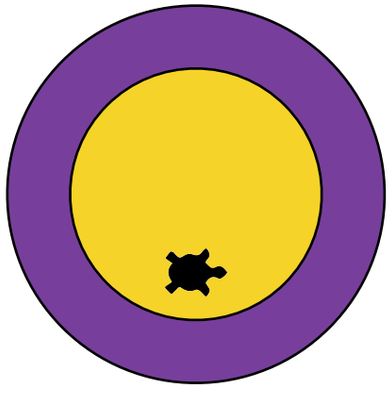
radius 3 gap 5

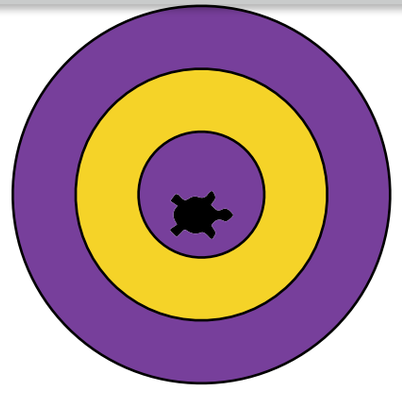
```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```

1

0







```
def concentric_circles(radius, gap, color_out, color_in):
  """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    draw_disc(radius, color_out)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(radius-gap, gap, color_in, color_out)
    return 1 + num
```

```
>>> concentric_circles(18, 5, "purple", "gold")
```

contrc_circles(18,5,'p','g')

radius 18 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = 2 ntric_circles
        g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(13,5,'g','p')

radius 13 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = 1 entric_circles
        -g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(8,5,'p','g')

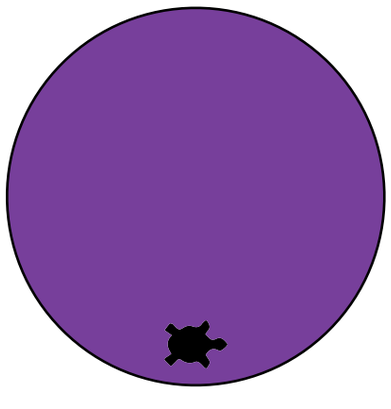
radius 8 gap 5

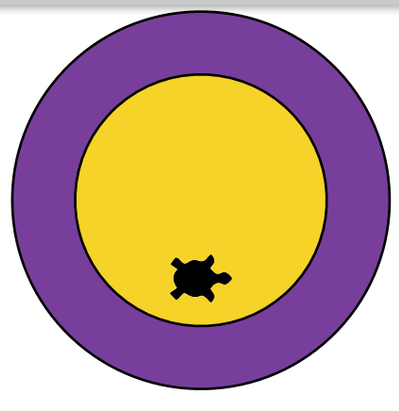
```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = c 0 tric_circles
        ( , g, clr_i, clr_o)
  return 1 + num
```

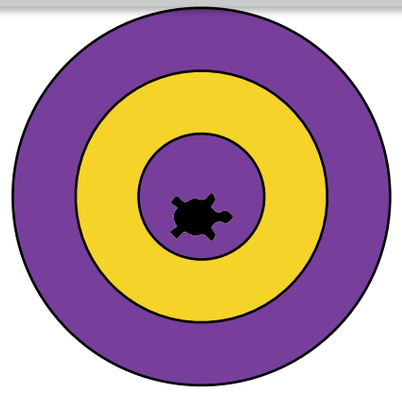
contrc_circles(3,5,'g','p')

radius 3 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```







```
def concentric_circles(radius, gap, color_out, color_in):
  """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    draw_disc(radius, color_out)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(radius-gap, gap, color_in, color_out)
    return 1 + num
```

```
>>> concentric_circles(18, 5, "purple", "gold")
```

contrc_circles(18,5,'p','g')

radius 18 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = 2 ntric_circles
        g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(13,5,'g','p')

radius 13 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = 1 entric_circles
        -g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(8,5,'p','g')

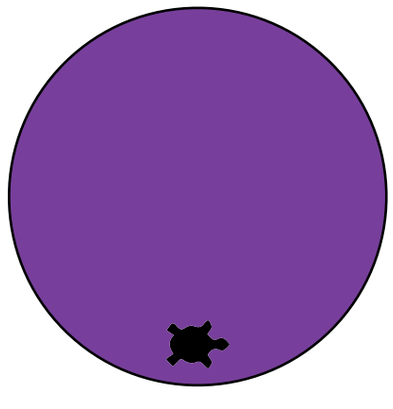
radius 8 gap 5

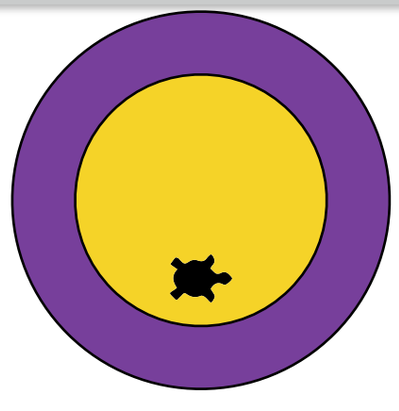
```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = c 0 tric_circles
        ( , g, clr_i, clr_o)
  return 1 + num
```

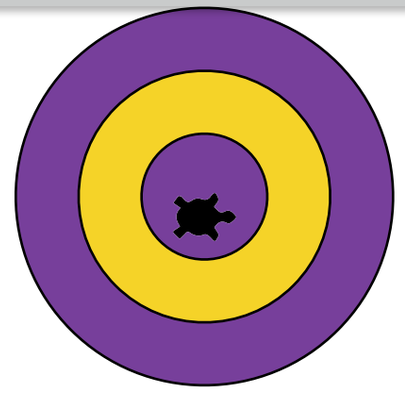
contrc_circles(3,5,'g','p')

radius 3 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```







```
def concentric_circles(radius, gap, color_out, color_in):
  """Recursive function to draw concentric circles"""
  if radius < gap:
    return 0
  else:
    draw_disc(radius, color_out)
    lt(90); fd(gap); rt(90)
    num = concentric_circles(radius-gap, gap, color_in, color_out)
    return 1 + num
```

```
>>> co 3 ntric_circles(18, 5, "purple", "gold")
```

contrc_circles(18,5,'p','g')

radius 18 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = 2 ntric_circles
        g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(13,5,'g','p')

radius 13 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = 1 entric_circles
        -g, g, clr_i, clr_o)
  return 1 + num
```

contrc_circles(8,5,'p','g')

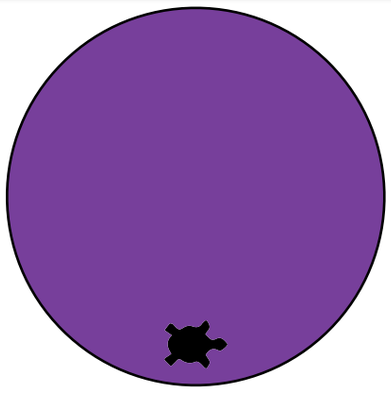
radius 8 gap 5

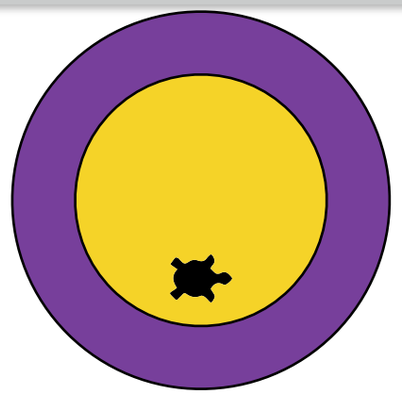
```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = c 0 tric_circles
        ( , g, clr_i, clr_o)
  return 1 + num
```

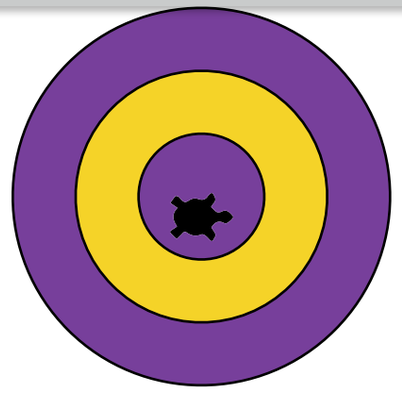
contrc_circles(3,5,'g','p')

radius 3 gap 5

```
if rad < gap:
  return 0
else:
  draw_disc(rad, clr_o)
  lt(90); fd(gap); rt(90)
  num = concentric_circles
        (rad-g, g, clr_i, clr_o)
  return 1 + num
```



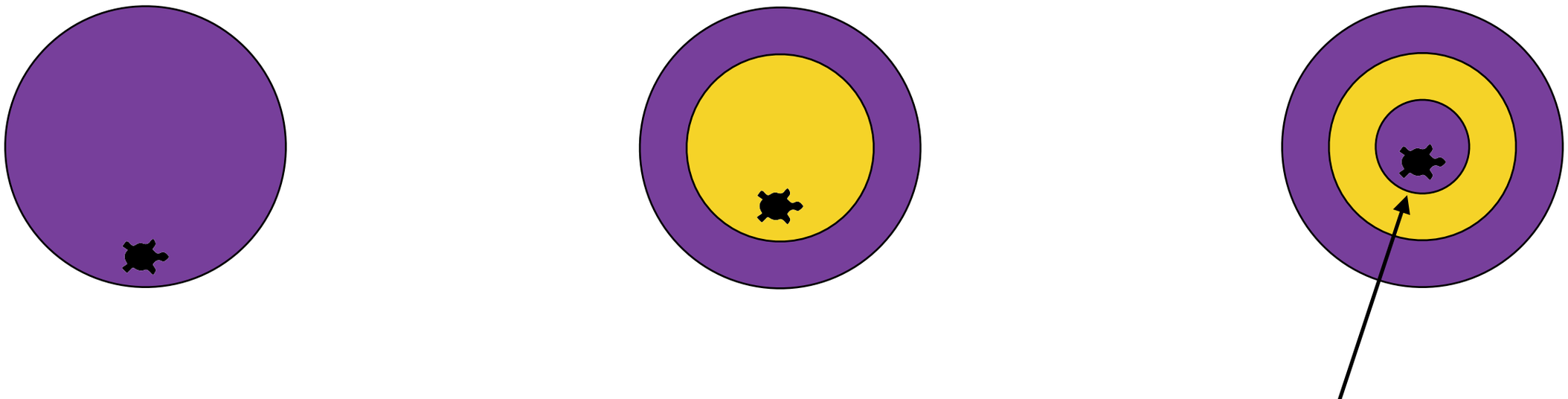




Invariance of Functions

- Where does the turtle end up in this example with `concentric_circles_color`?

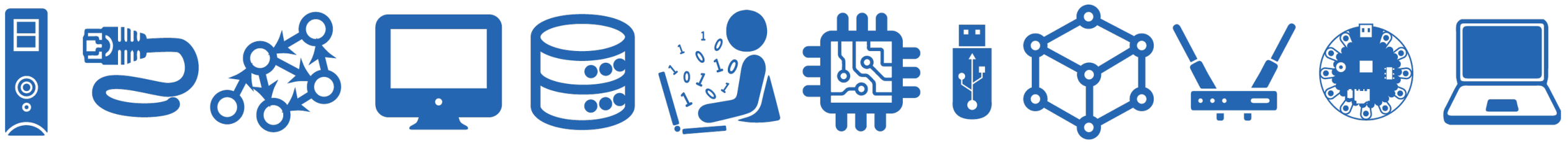
```
concentric_circles(18, 5, 'purple', 'gold')
```



- The turtle does not end where it starts

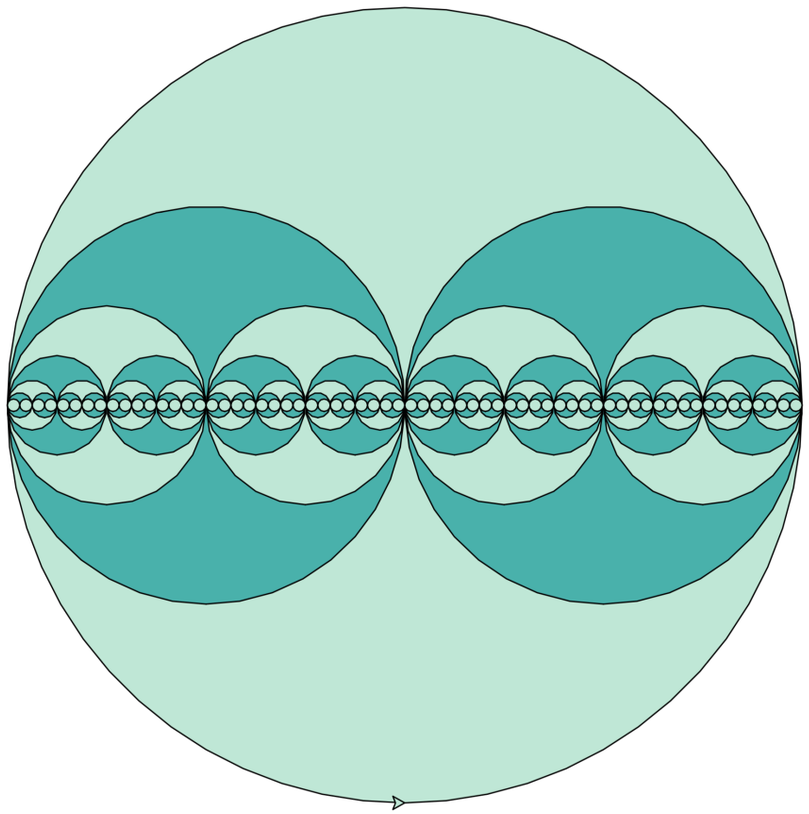
```
def concentric_circles_color(radius, gap, color_outer, color_inner):
    """
    Recursive function to draw concentric circles with
    alternating colors
    """
    # base case, don't draw anything, return 0
    if radius < gap:
        return 0
    else:
        draw_disc(radius, color_outer)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

Example: Nested Circles



Invariance of Recursive Functions

- Let's do an example with multiple recursive calls: nested circles

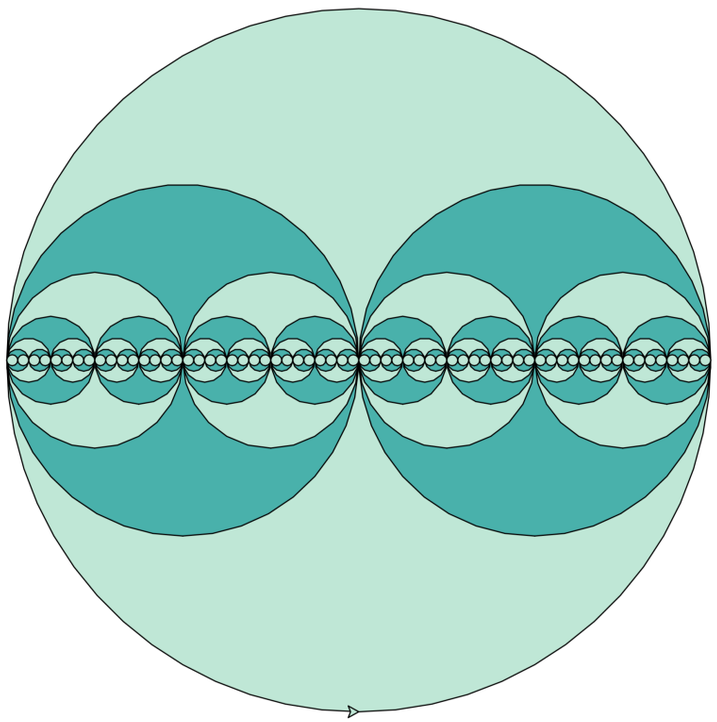


Multiple Recursive Calls

- **Example:** Nested circles function definition

```
nested_circles(radius, min_radius, color_out, color_alt)
```

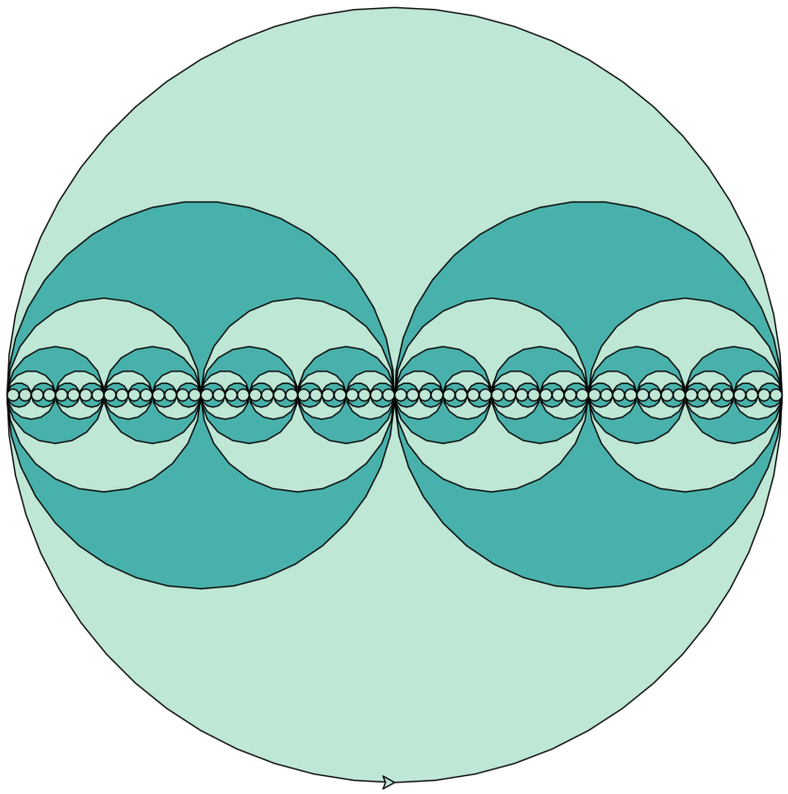
  - `radius`: radius of the outermost circle
  - `min_radius`: minimum radius of any circle
  - `color_out`: color of the outermost circle
  - `color_alt`: color that alternates with colorOut

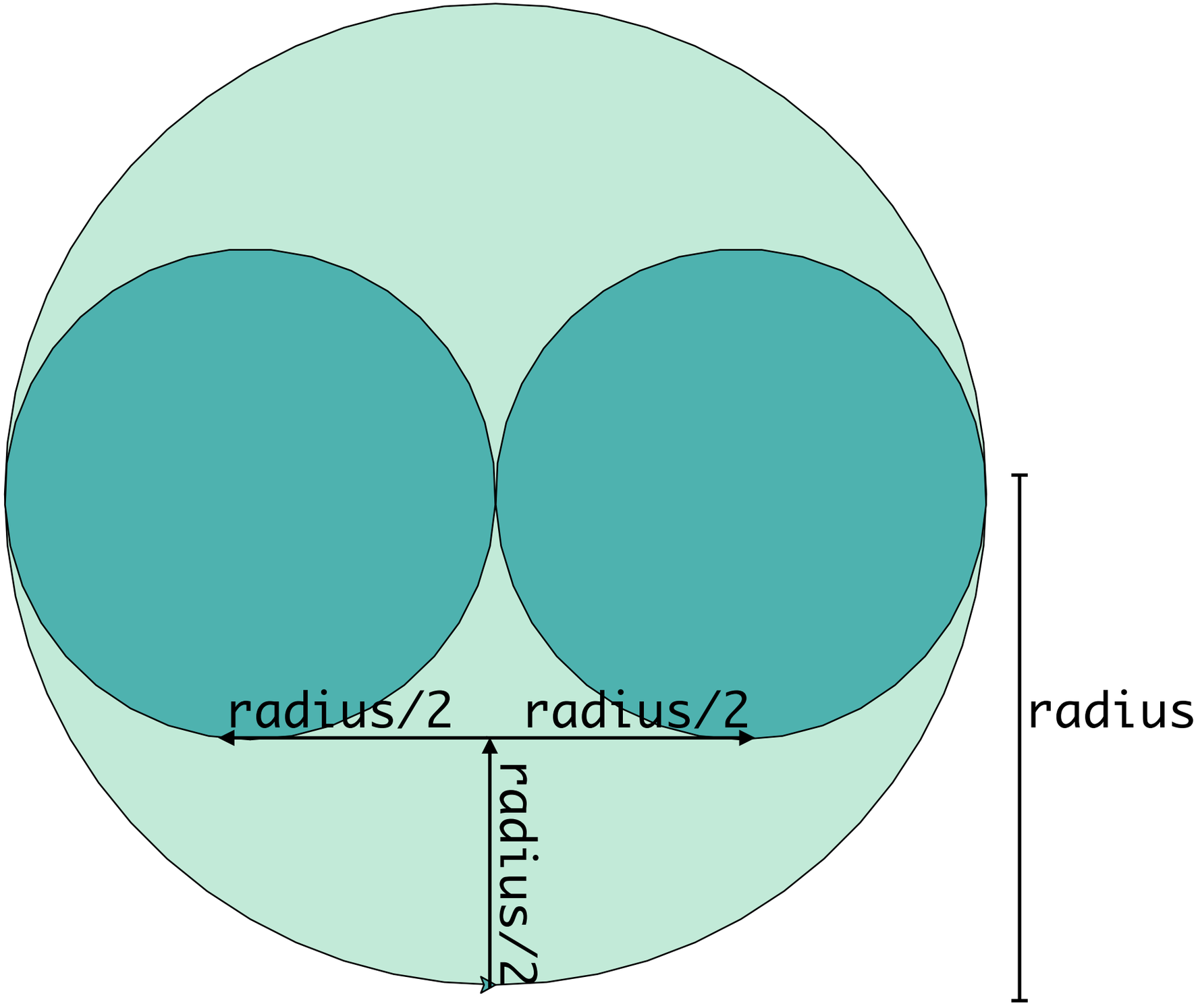


Nested Circles

- **Base case?**
  - When radius becomes less than min_radius
  - Don't draw anything return 0

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



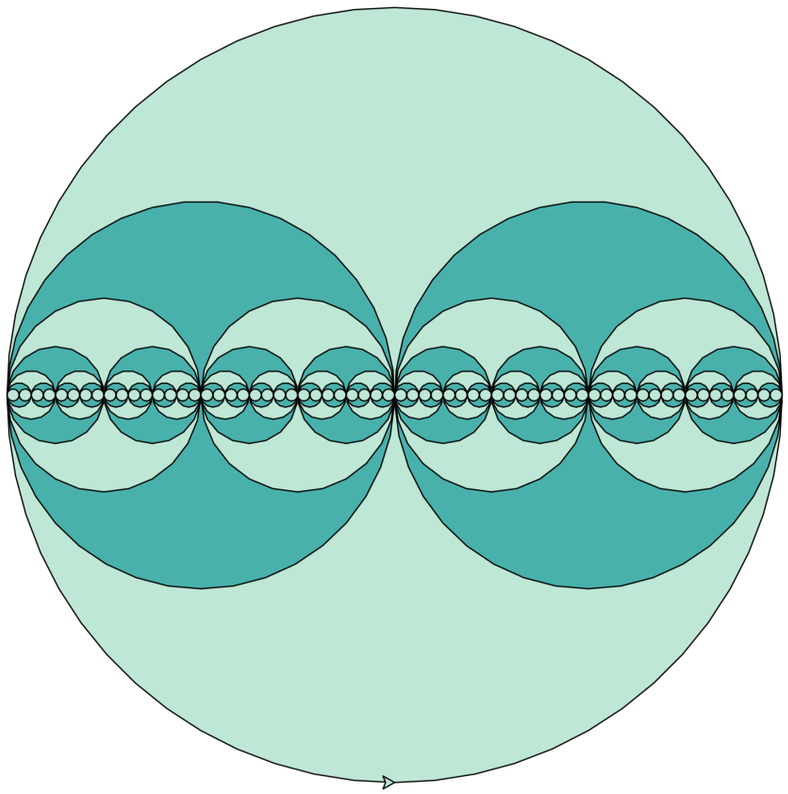


Starting position of turtle

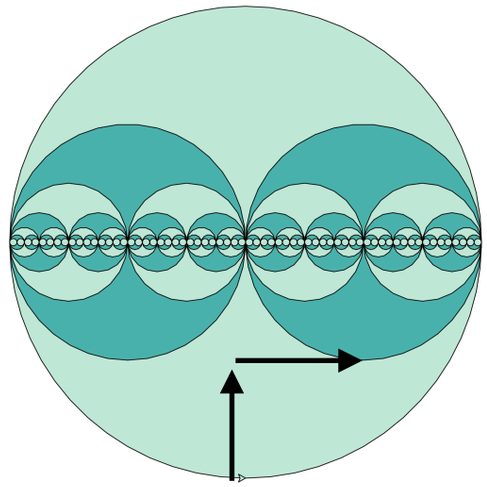
```
nested_circles(300, 150)
```

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



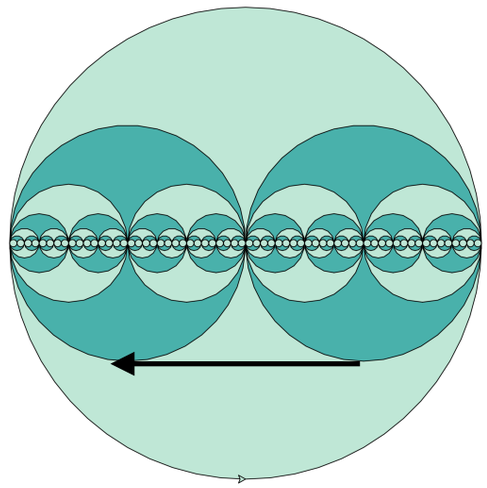
Nested Circles



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

```
def nested_circles(radius, min_radius, color_out, color_alt):
    if radius < min_radius:
        return 0
    else:
        # contribute to the solution
        draw_disc(radius, color_out)

        # save half of radius
        half_radius = radius/2

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

        # draw right subcircle recursively
        right = nested_circles(half_radius, min_radius, color_alt, color_out)
```

Nested Circles



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

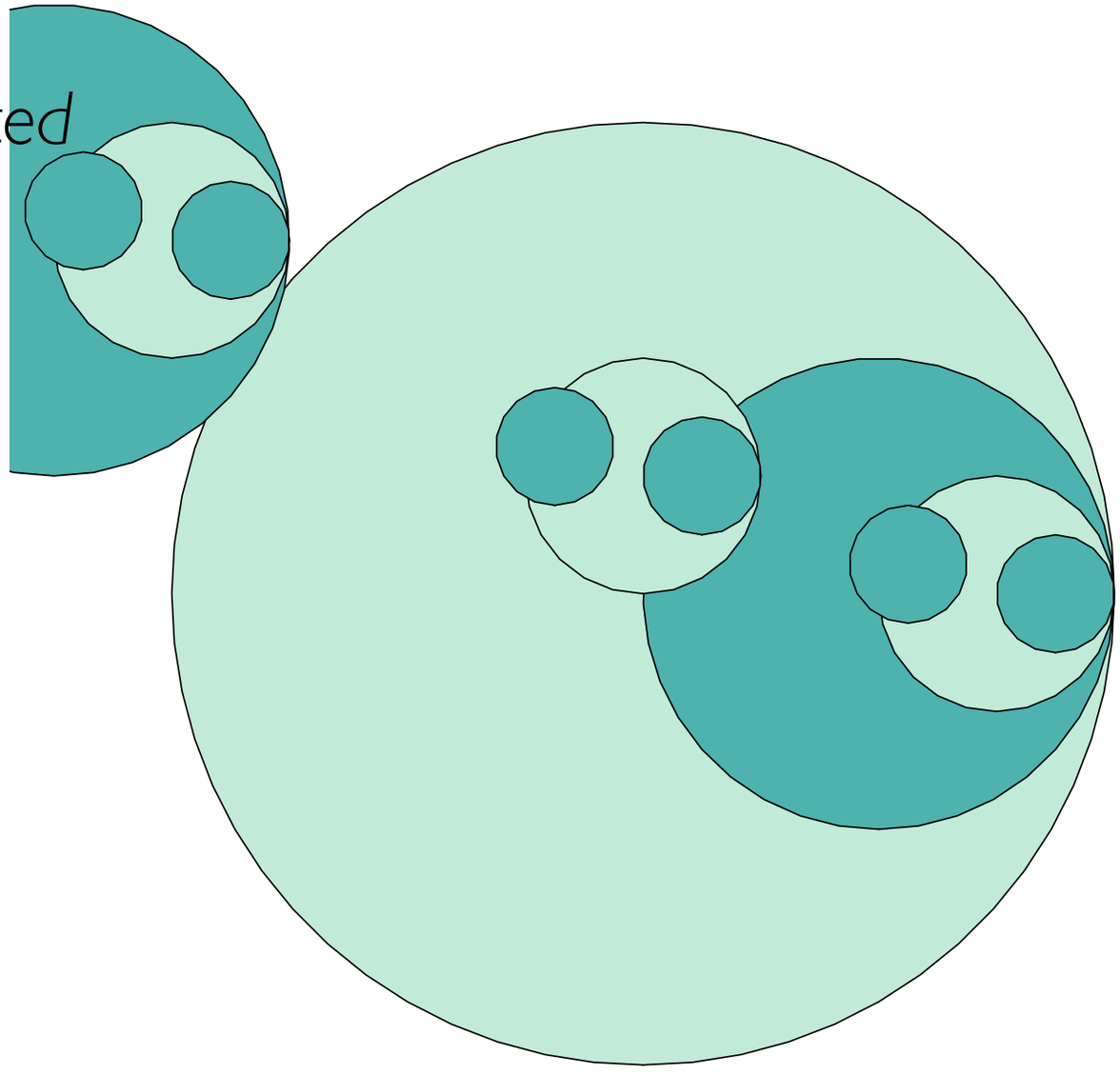
```
        # draw right subcircle recursively
        right = nested_circles(half_radius, min_radius, color_alt, color_out)

        # position turtle for left subcircle
        bk(radius)

        # draw left subcircle recursively
        left = nested_circles(half_radius, min_radius, color_alt, color_out)

        # add one to our count of subcircles
        return 1 + left + right
```

Nested Circles

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

Nested Circles

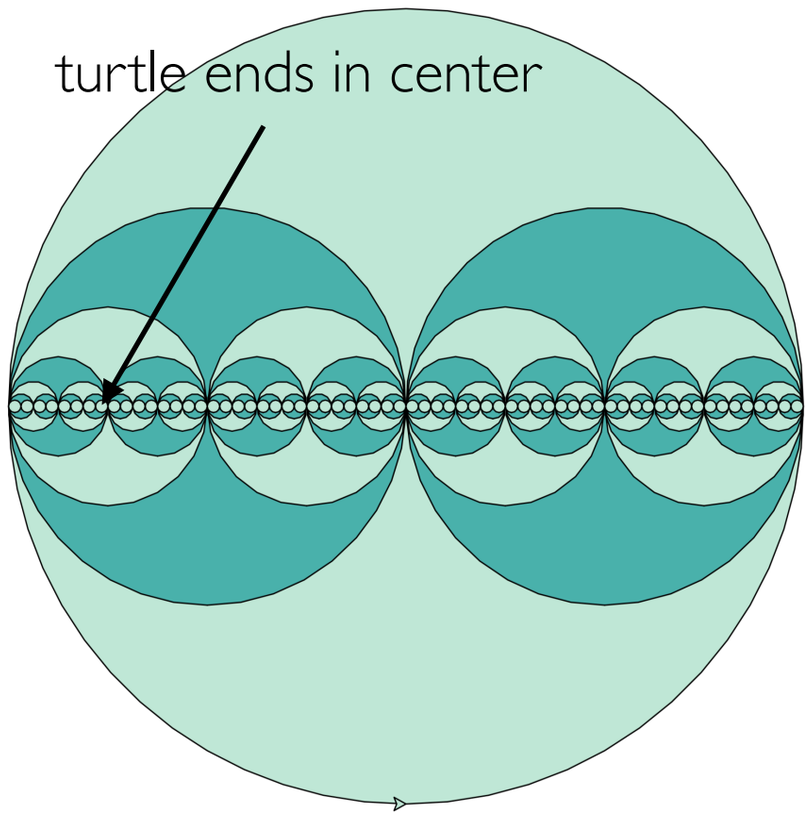
- **Recursive case**
  - What happened?!
  - We made assumptions about where the turtle started, that wasn't true!
  - Need turtle to *end* where it *started*



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 `nested_circles` 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?

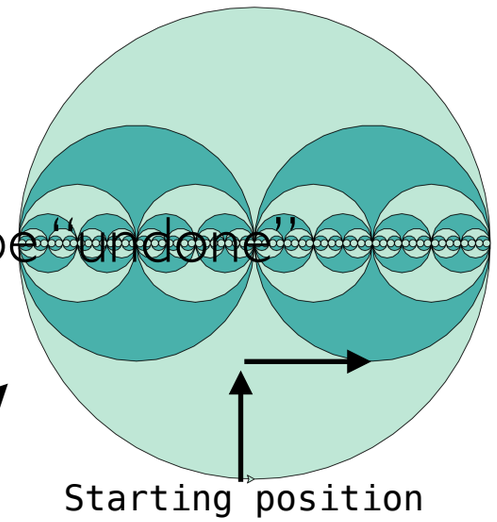
```
def nested_circles(radius, min_radius, color_out, color_alt):
    if radius < min_radius:
        return 0
    else:
        draw_disc(radius, color_out)
        h_r = radius/2

        lt(90); fd(h_r); rt(90); fd(h_r)

        right = nested_circles(h_r, min_radius, color_alt, color_out)

        bk(radius)

        left = nested_circles(h_r, min_radius, color_alt, color_out)

        fd(h_r);  lt(90);  bk(h_r);  rt(90)
        return 1 + right + left
```

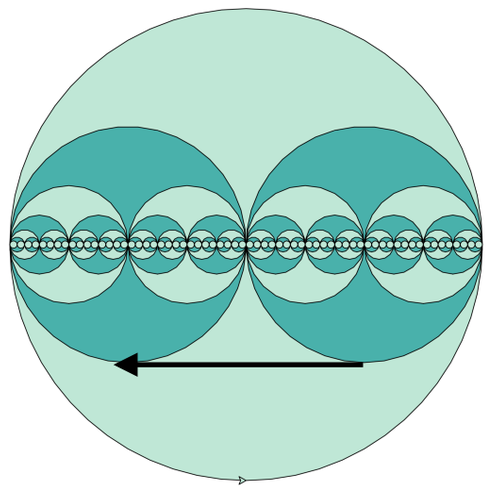


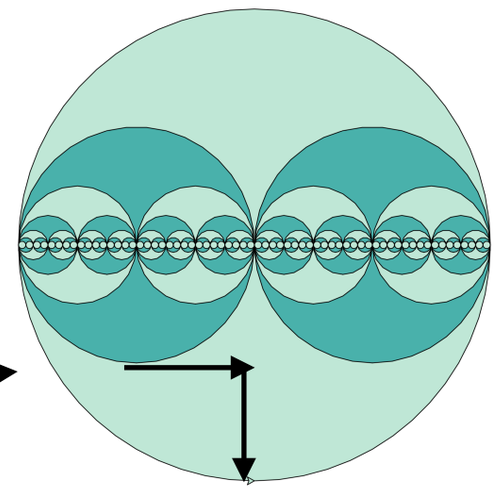
Maintaining Invariance

- 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 nested_circles(radius, min_radius, color_out, color_alt):
    if radius < min_radius:
        return 0
    else:
        # contribute to the solution
        draw_disc(radius, color_out)

    # save half of radius
    half_radius = radius/2

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

    # draw right subcircle recursively
    right = nested_circles(half_radius, min_radius, color_alt, color_out)

    # position turtle for left subcircle
    bk(radius)

    # draw left subcircle recursively
    left = nested_circles(half_radius, min_radius, color_alt, color_out)

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

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

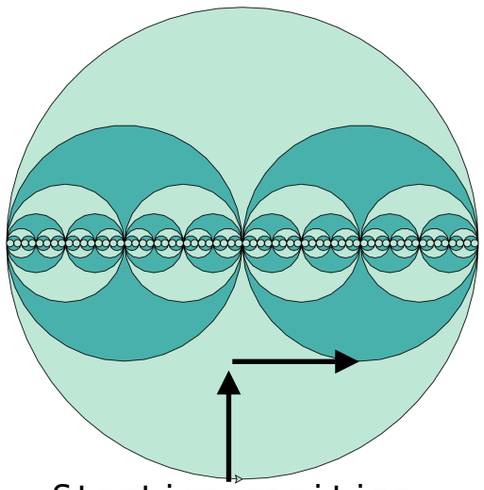




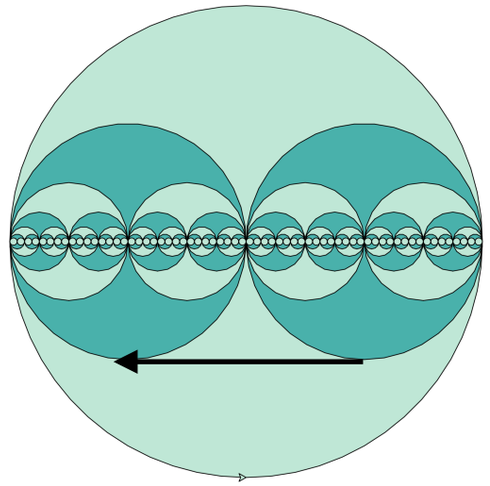


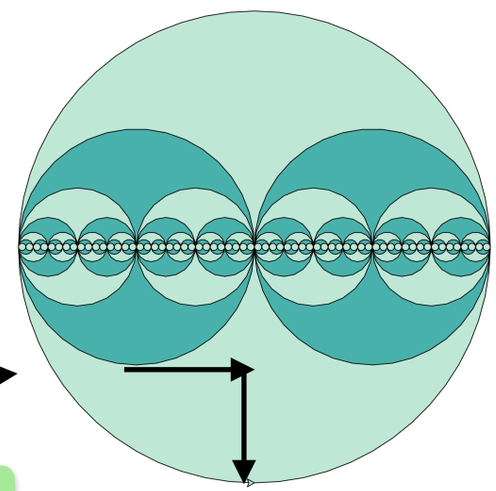
Maintaining Invariance

- Move turtle back to starting position to maintain **invariance**

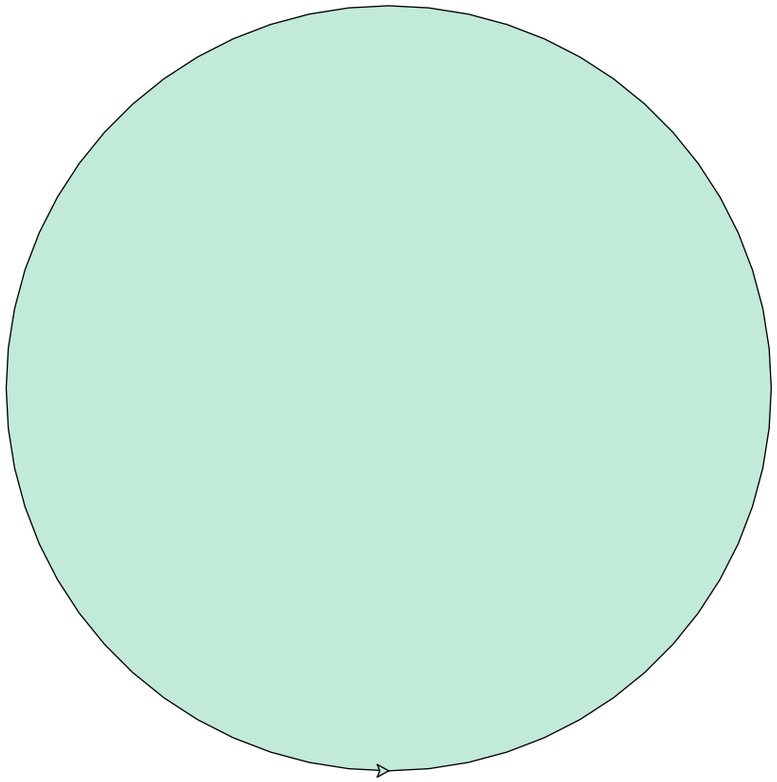



```
def nested_circles(radius, min_radius, color_out, color_alt):
    if radius < min_radius:
        return 0
    else:
        # contribute to the solution
        draw_disc(radius, color_out)

    # save half of radius
    half_radius = radius/2

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

    # draw right subcircle recursively
    right = nested_circles(half_radius, min_radius, color_alt, color_out)

    # position turtle for left subcircle
    bk(radius)

    # draw left subcircle recursively
    left = nested_circles(half_radius, min_radius, color_alt, color_out)

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

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

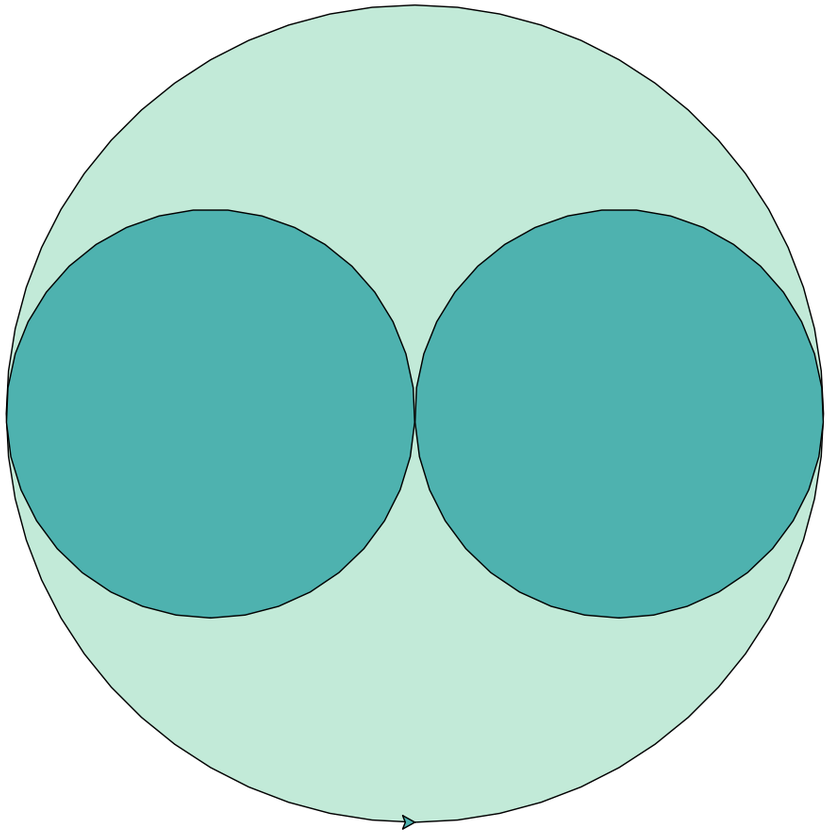




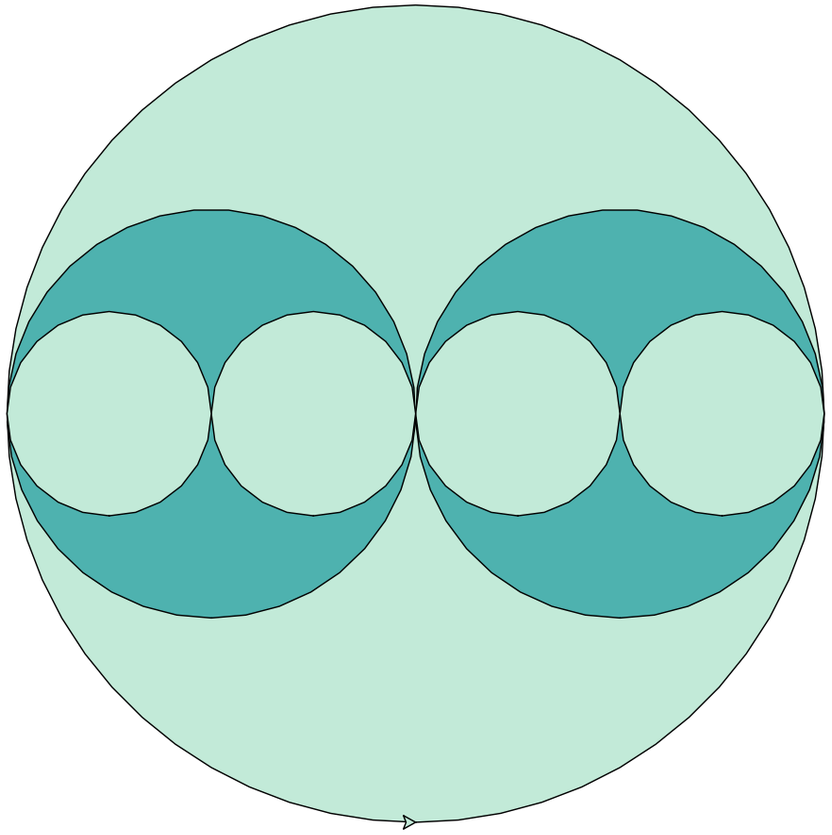
Maintain invariance



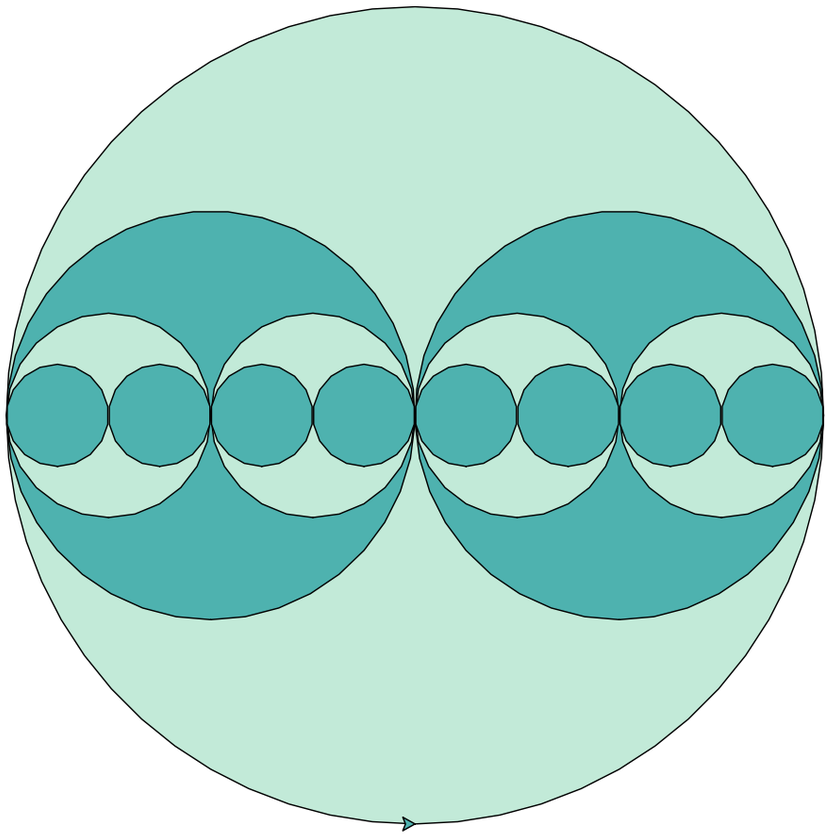
`nestedCircles(300, 300)`



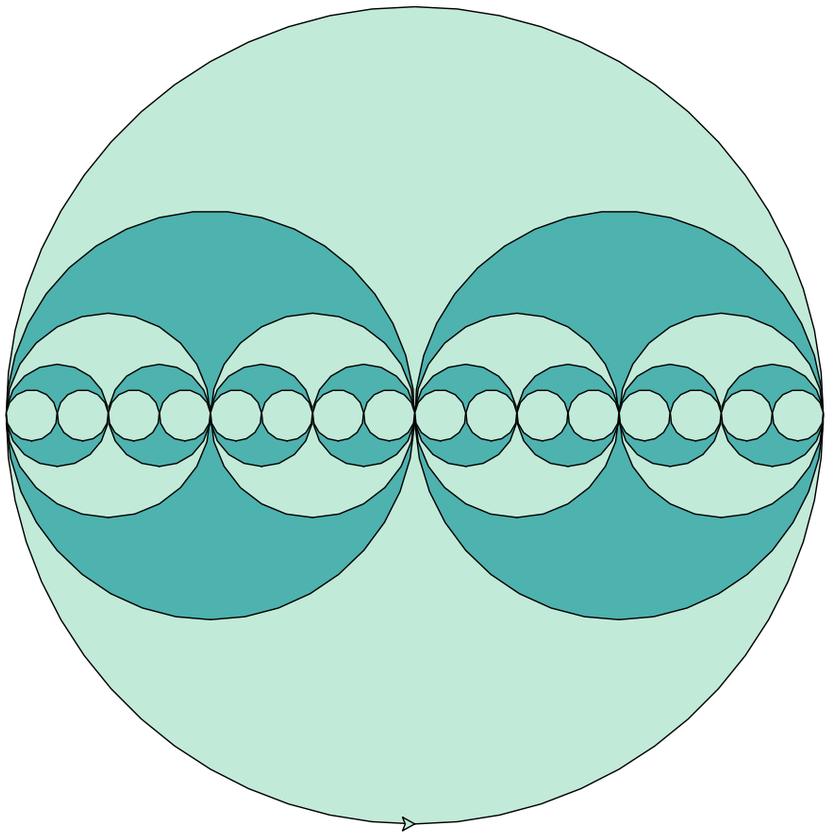
`nestedCircles(300, 150)`



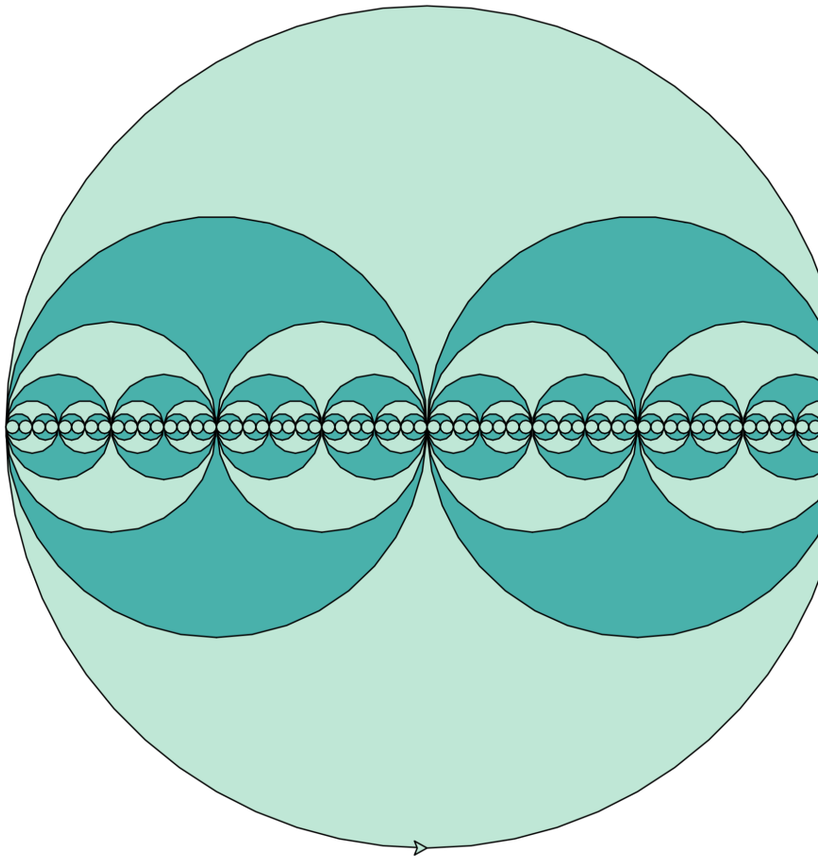
`nestedCircles(300, 75)`



`nestedCircles(300, 37)`



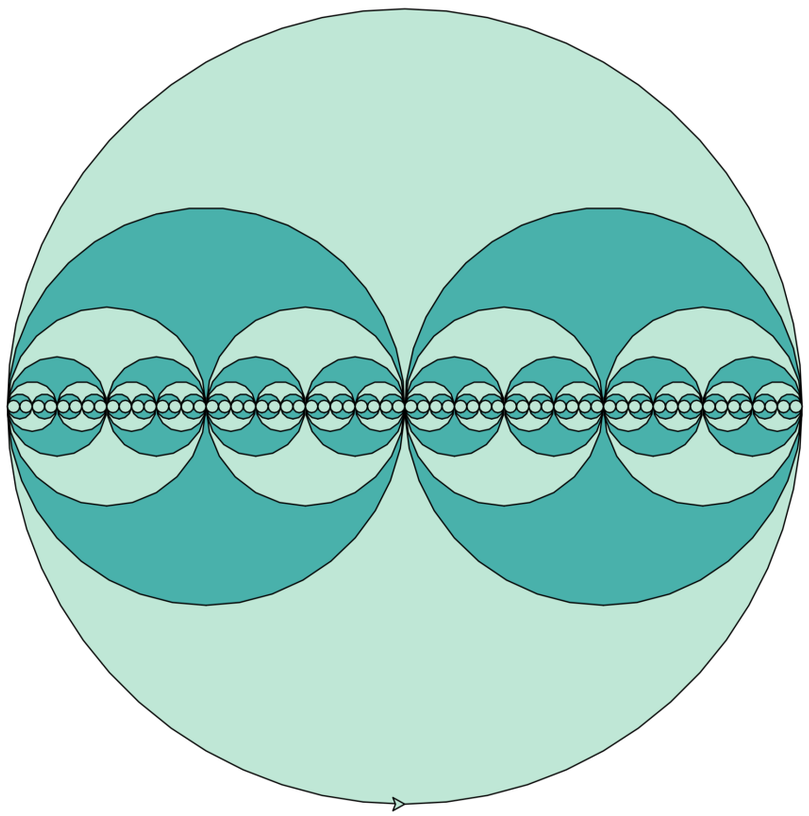
`nestedCircles(300, 9)`



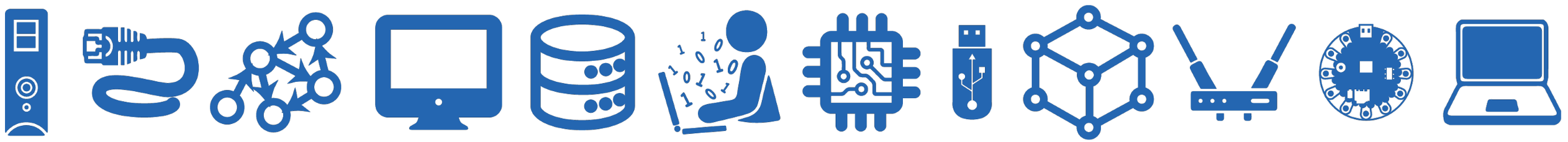
`nestedCircles(300, 2)`

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



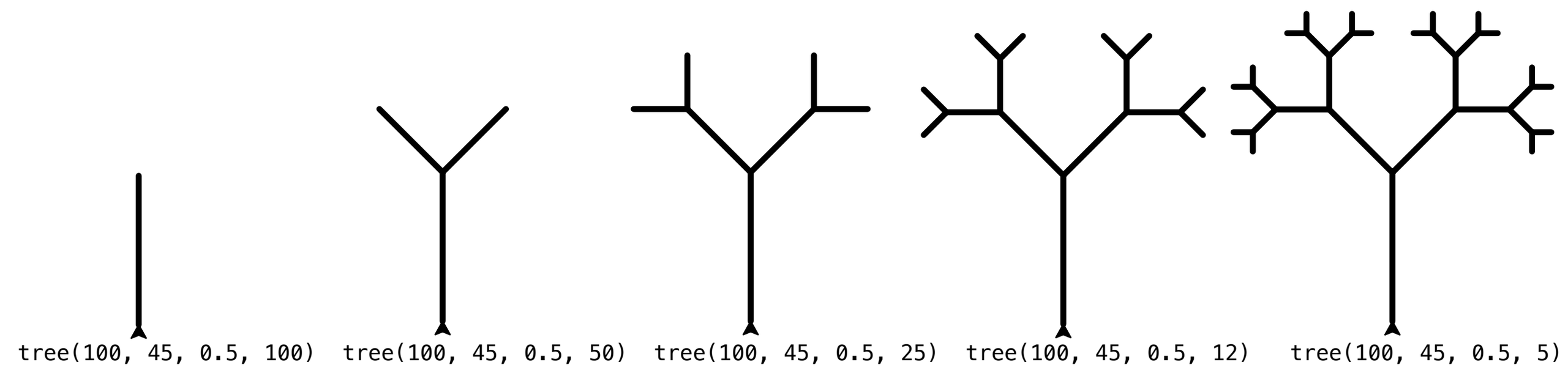
Recursive Trees



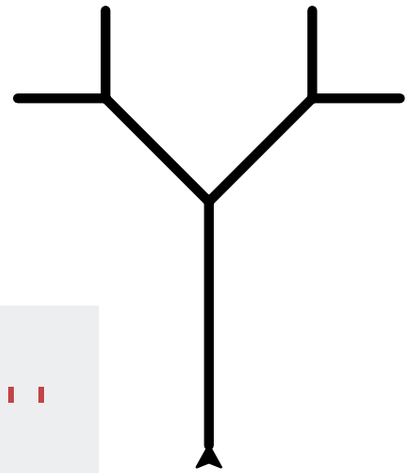
One more recursive example: Trees

- We can draw more than just circles!
- Suppose we want to draw recursive trees and count branches
- What is our base case? Recursive case?
- *Note:* Assume turtle starts facing north

```
def tree(trunk_len, angle, shrink_factor, min_len):
    # trunk_len is the trunk length of the main (vertical) trunk
    # angle is the branching angle, or the angle between a trunk and its
      right or left branch
    # shrink_factor specifies how much smaller each subsequent branch is in length
    # min_len is the minimum branch length in our tree
```

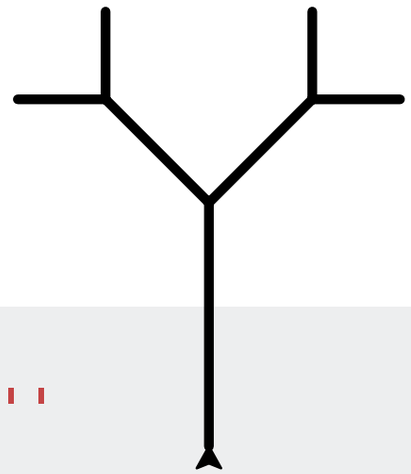


Tree



```
def tree(trunk_len, angle, shrink_factor, min_len):
    '''Draw tree and return number of branches drawn including trunk'''
   # Base case: trunk_len < min_len
        # return 0, don't draw anything!
   # Recursive case:
        # Draw trunk



        # Position for Right branch: Turn right angle
        # Right branch -> shrink trunk, pass along other variables


    # Position for Left branch: Turn left angle*2
    # Left branch -> shrink trunk, pass along other variables


    # Maintain invariance
    # Turn right, then back up to starting position

    # return 1 (for the trunk we drew), plus the sum of the branches
```

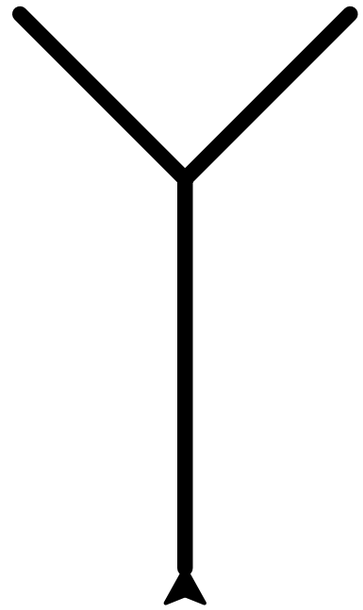
Tree



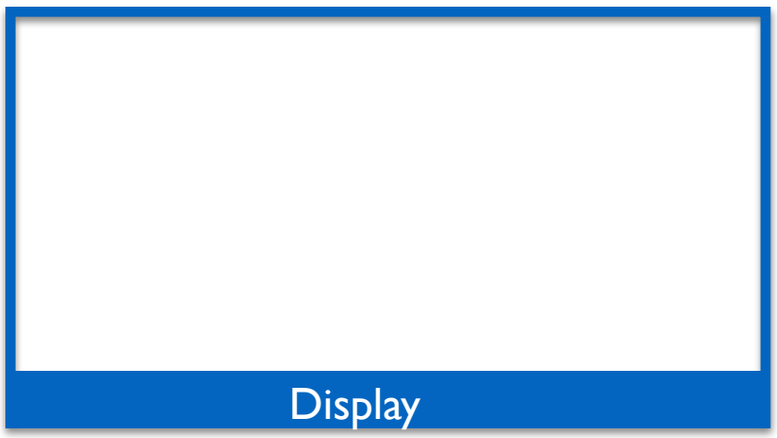
```
def tree(trunk_len, angle, shrink_factor, min_len):
    '''Draw tree and return number of branches drawn including trunk'''
    if (trunk_len < min_len): # Base case
        return 0
    else:
        # Draw trunk
        fd(trunk_len)

        # Right branch
        rt(angle)
        right_branch = tree(trunk_len*shrink_factor, angle, shrink_factor, min_len)

        # Left branch
        lt(angle*2)
        left_branch = tree(trunk_len*shrink_factor, angle, shrink_factor, min_len)

        # Maintain invariance
        rt(angle); bk(trunk_len)

        return 1 + right_branch + left_branch
```

Function Frame Model: `tree`

```
def tree(trunk_len, angle, shrink_factor, min_len):
    '''Draw tree and return number of branches drawn including trunk'''
    if (trunk_len < min_len): # Base case
        return 0
    else:
        fd(trunk_len)
        rt(angle)
        right_br = tree(trunk_len*shrink_factor, angle, shrink_factor, min_len)
        lt(angle*2)
        left_br = tree(trunk_len*shrink_factor, angle, shrink_factor, min_len)
        rt(angle); bk(trunk_len)
        return 1 + right_br + left_br
```



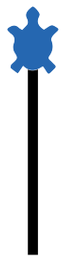
```
>>> tree(100, 45, 0.5, 50)
```



```
tree(100, 45, 0.5, 50)
```

trunk_len **100** min **50**

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```



Display

tree(100, 45, 0.5, 50)

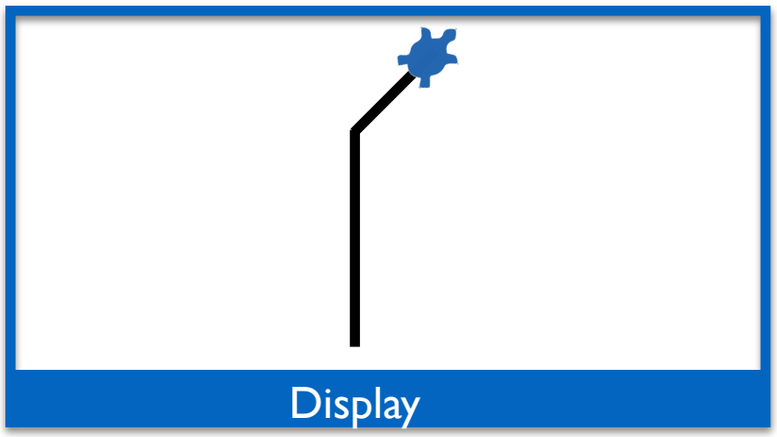
trunk_len 100 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```



tree(100, 45, 0.5, 50)

trunk_len 100 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(50, 45, 0.5, 50)

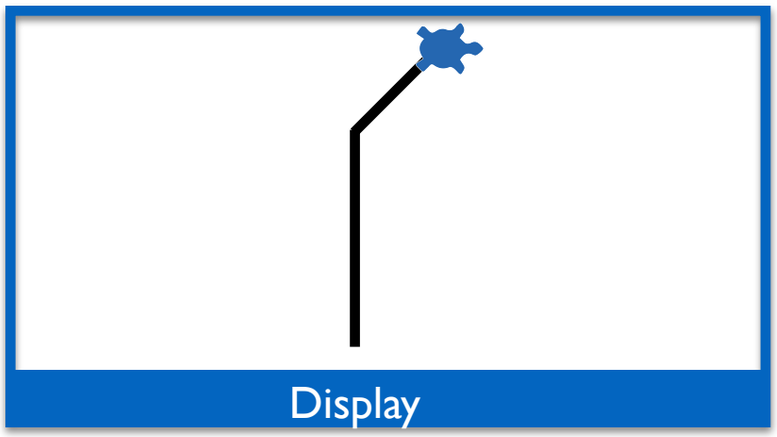
trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```



tree(100, 45, 0.5, 50)

trunk_len **100** min **50**

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(50, 45, 0.5, 50)

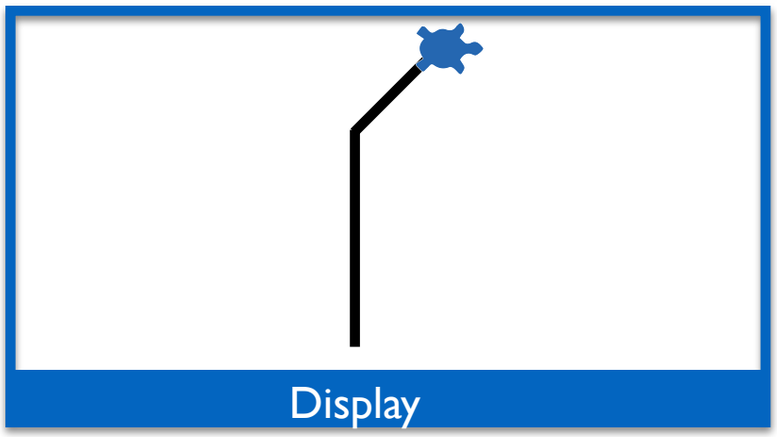
trunk_len **50** min **50**

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree 0 *shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len **25** min **50**

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```



tree(100, 45, 0.5, 50)

trunk_len 100 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(50, 45, 0.5, 50)

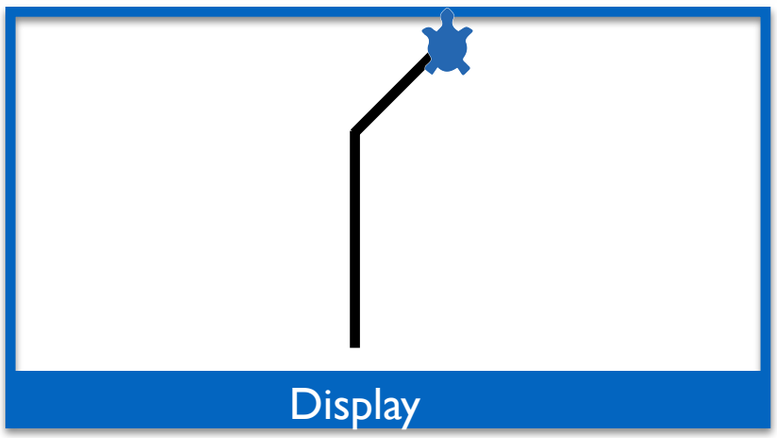
trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree 0 *shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```



tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(100, 45, 0.5, 50)

trunk_len **100** min **50**

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(50, 45, 0.5, 50)

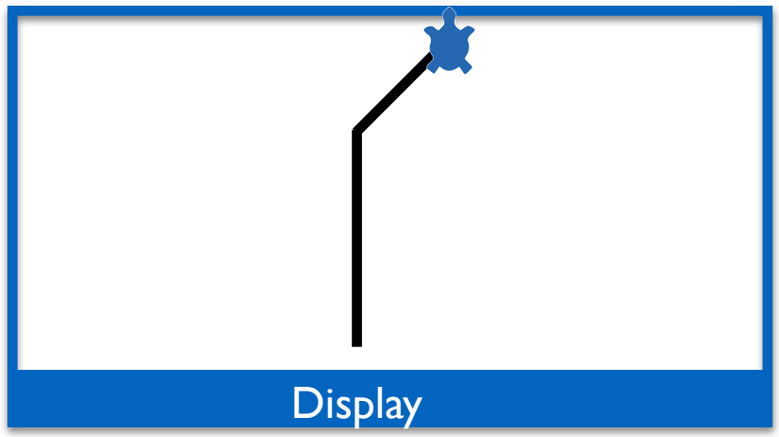
trunk_len **50** min **50**

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree 0 (*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrin 0 )
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len **25** min **50**

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```



tree(25, 45, 0.5, 50)

trunk_len **25** min **50**

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(100, 45, 0.5, 50)

trunk_len 100 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(50, 45, 0.5, 50)

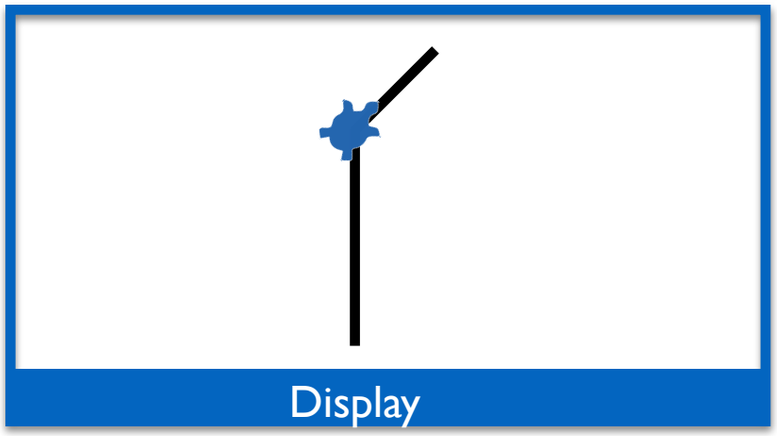
trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree 0 (*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrin 0 )
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```



tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(100, 45, 0.5, 50)

trunk_len **100** min **50**

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

1

tree(50, 45, 0.5, 50)

trunk_len **50** min **50**

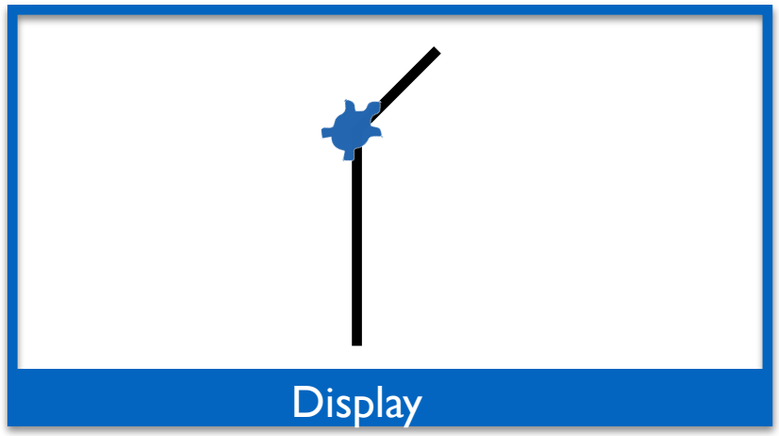
```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

0 0

tree(25, 45, 0.5, 50)

trunk_len **25** min **50**

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```



Display

tree(25, 45, 0.5, 50)

trunk_len **25** min **50**

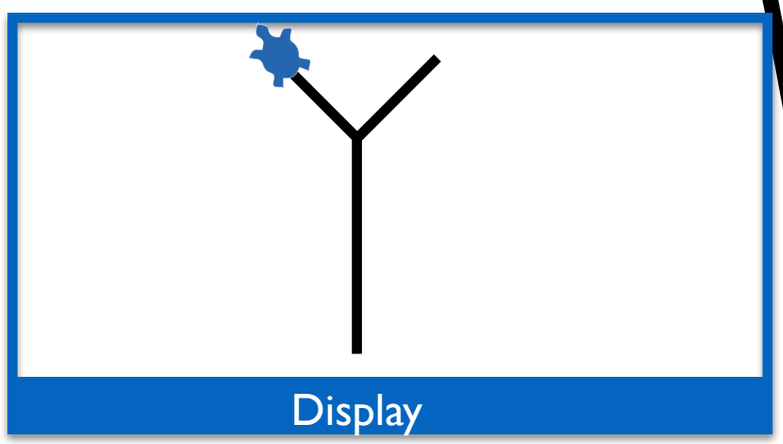
```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(100, 45, 0.5, 50)

trunk_len 100 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

1



tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

0

0

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(100, 45, 0.5, 50)

trunk_len 100 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

1

tree(50, 45, 0.5, 50)

trunk_len 50 min 50

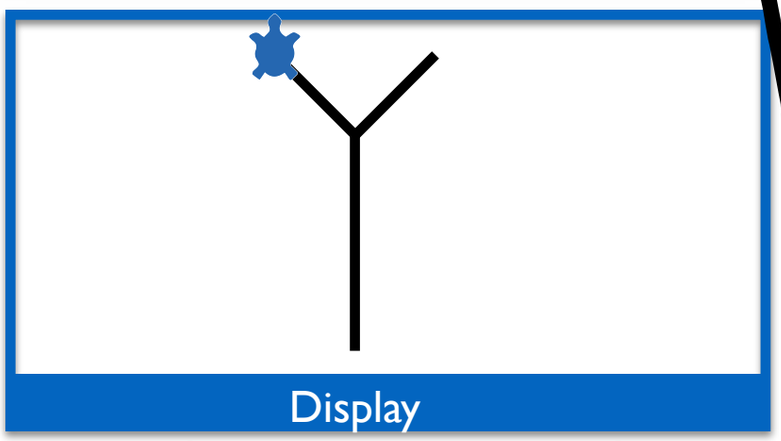
```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

0 0

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```



Display

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(100, 45, 0.5, 50)

trunk_len 100 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

1

tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

0 0

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

Display

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

0

tree(25, 45, 0.5, 50)

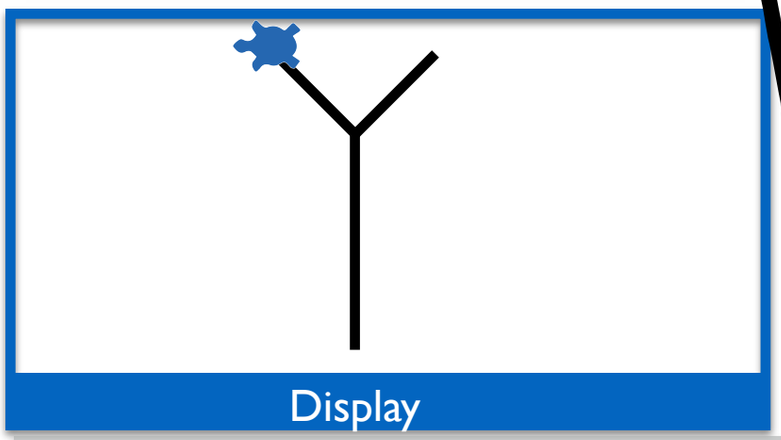
trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(100, 45, 0.5, 50)

trunk_len 100 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)   # 1
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```



Display

tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)   # 0
      lt(angle*2)
      left = tree(trunk*shrink,...)   # 0
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
...
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)   # 0
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

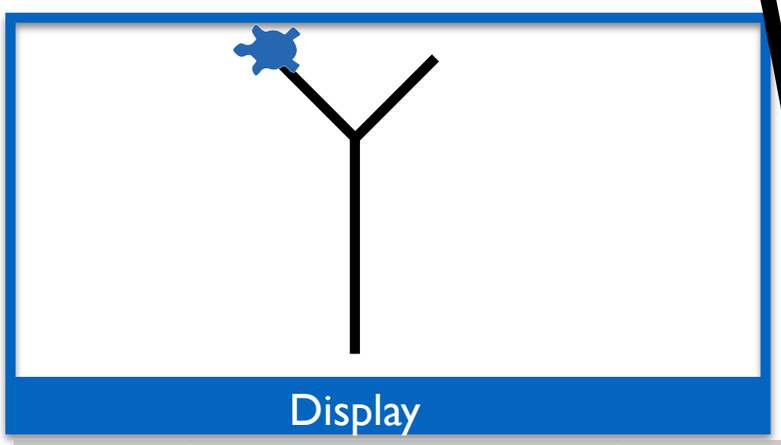
trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(100, 45, 0.5, 50)

trunk_len 100 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)   1
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```



Display

tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)   0
      lt(angle*2)
      left = tree(trunk*shrink,...)   0
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
...
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)   0
      lt(angle*2)
      left = tree(trunk*shrink,...)   0
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

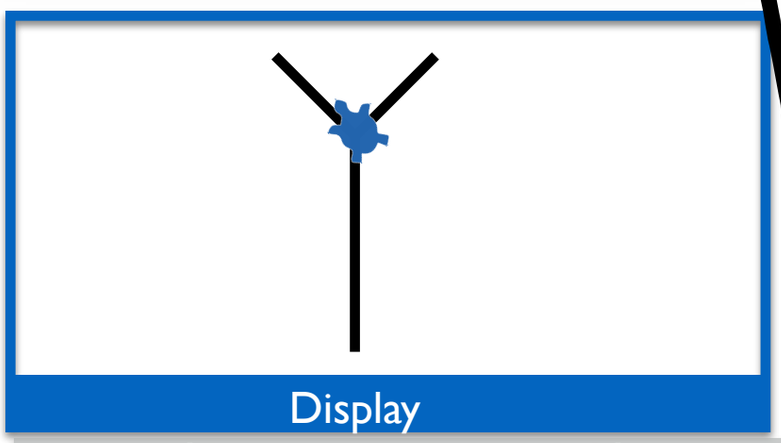
trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(100, 45, 0.5, 50)

trunk_len 100 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)   # 1
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```



Display

tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)   # 0
      lt(angle*2)
      left = tree(trunk*shrink,...)   # 0
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
...
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)   # 0
      lt(angle*2)
      left = tree(trunk*shrink,...)   # 0
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

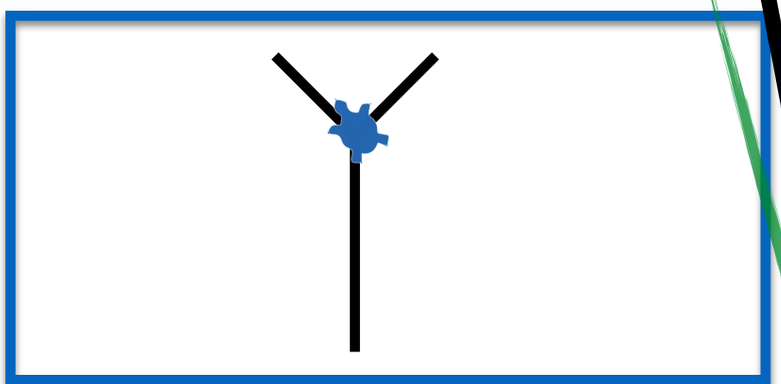
trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(100, 45, 0.5, 50)

trunk_len 100 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)   # 1
      lt(angle*2)
      left = tree(trunk*shrink,...)   # 1
      rt(angle); bk(trunk)
      return 1 + right + left
```



tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)   # 0
      lt(angle*2)
      left = tree(trunk*shrink,...)   # 0
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
...
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)   # 0
      lt(angle*2)
      left = tree(trunk*shrink,...)   # 0
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

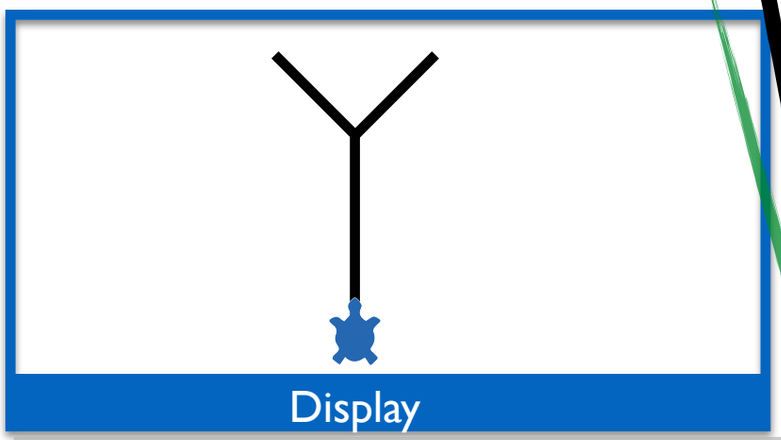
trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(100, 45, 0.5, 50)

trunk_len 100 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)   # 1
      lt(angle*2)
      left = tree(trunk*shrink,...)   # 1
      rt(angle); bk(trunk)
      return 1 + right + left
```



tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)   # 0
      lt(angle*2)
      left = tree(trunk*shrink,...)   # 0
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
...
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)   # 0
      lt(angle*2)
      left = tree(trunk*shrink,...)   # 0
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(100, 45, 0.5, 50)

trunk_len 100 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

1

1

tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

0

0

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
...
```

3

Display

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

tree(50, 45, 0.5, 50)

trunk_len 50 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

0

0

tree(25, 45, 0.5, 50)

trunk_len 25 min 50

```
def tree(trunk, ang, shrink, min):
    if (trunk < min):
      return 0
    else:
      fd(trunk)
      rt(angle)
      right = tree(trunk*shrink,..)
      lt(angle*2)
      left = tree(trunk*shrink,...)
      rt(angle); bk(trunk)
      return 1 + right + left
```

Recursion: Wrap Up

- Why choose recursion over iteration?
  - Some problems have a ***natural recursive structure***
  - Using recursion on them leads to elegant and concise solutions
  - Fewer lines of code often correlates with less debugging!
- We will use recursion to search and sort in a few weeks
- Recursion also helps us build and maintain complex data structures
- Downsides: Recursive approaches *can* have efficiency overhead
    - Steeper learning curve (but can be very rewarding once you get the hang of it)
    - To understand recursion you must understand recursion…

The end!