Fruitful and Graphical Recursion

Recursive Algorithm

- Base case: Solving problem directly.
- Recursive case:
    - REDUCE the problem to smaller subproblem(s) (smaller version(s) of itself)
    - DELEGATE the smaller problems to the recursion fairy *(formally known as induction hypothesis)* and assume they're solved correctly
    - COMBINE the solution(s) of the smaller subproblems to reach/return the solution

Fruitful Recursion

- We say a recursion is fruitful if the recursive function returns a value (other than `None`)

sumUp(n)

- Let's write a fruitful recursive function that sums up integers from `1` down to `n` (**without loops**)
- Recursive case. (REDUCE/ DELEGATE/ COMBINE): Can think of `sum(5)` as `5 + sumUp(4)`

```
In[1] sumUp(5)
Out[1] 15
```

```
In[2] sumUP(10)
Out[2] 55
```

Unfolding the Recursion

```
def sumUp(n):
    """returns sum of integers
    from 1 up to n"""
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```

```
sumUp(4)
⇒ 4 + sumUp(3)
⇒ 4 + (3 + sumUp(2))
⇒ 4 + (3 + sumUp(2))
⇒ 4 + (3 + (2 + sumUp(1)))
⇒ 4 + (3 + (2 + (1 + sumUp(0))))
⇒ 4 + (3 + (2 + (1 + 0)))
⇒ 4 + (3 + (2 + 1))
⇒ 4 + (3 + 3)
⇒ 4 + 6
⇒ 10
```

Fruitful Recursion: Base Case(s) Required!

```
def countUp(n):
    if n <= 0:
        pass
    else:
        countUp(n-1)
        print(n)
```

```
def countUp(n):
    if n > 0:
        countUp(n-1)
        print(n)
```

```
def sumUp(n):
    if n <= 0:
        return 0
    else:
        return n + sumUp(n-1)
```

What happens if we eliminate the base case for sumUp?

Palindromes

EVE

CIVIC

MADAM

AVID DIVA

STEP ON NO PETS

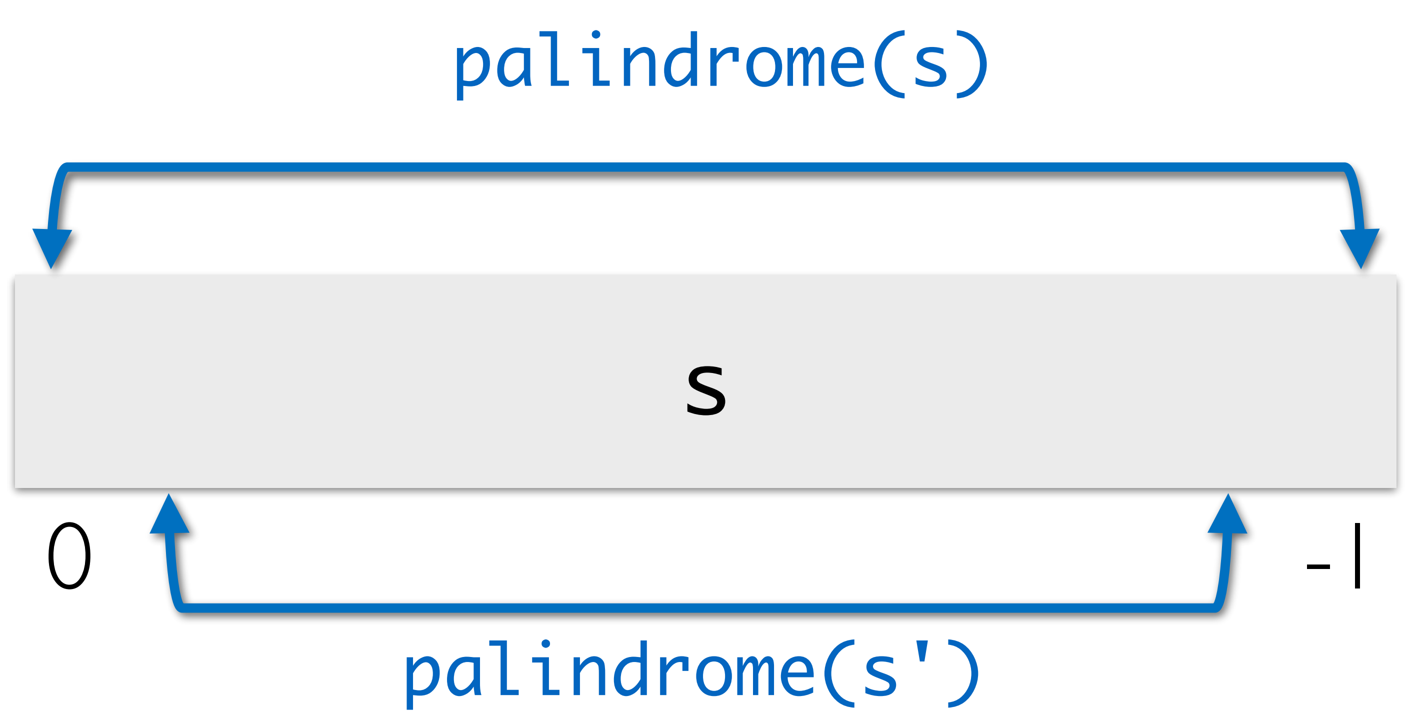
STRESSED DESSERTS

ABLE WAS I ERE I SAW ELBA

LIVED ON DECAF FACED NO DEVIL

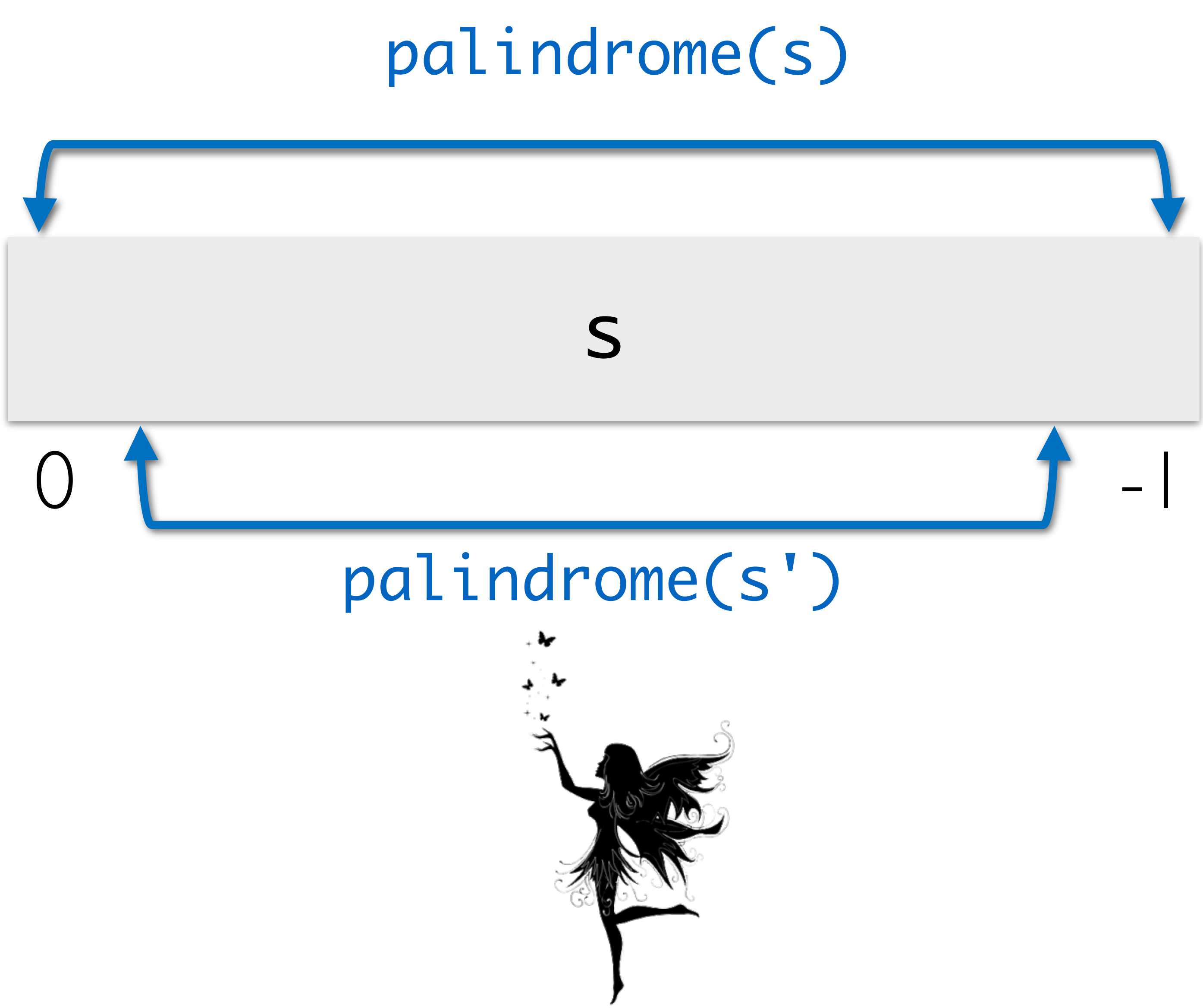
Recursive Approach

- REDUCE it smaller version of the same problem
  - Check if `s' = s[1:-1]` is a palindrome



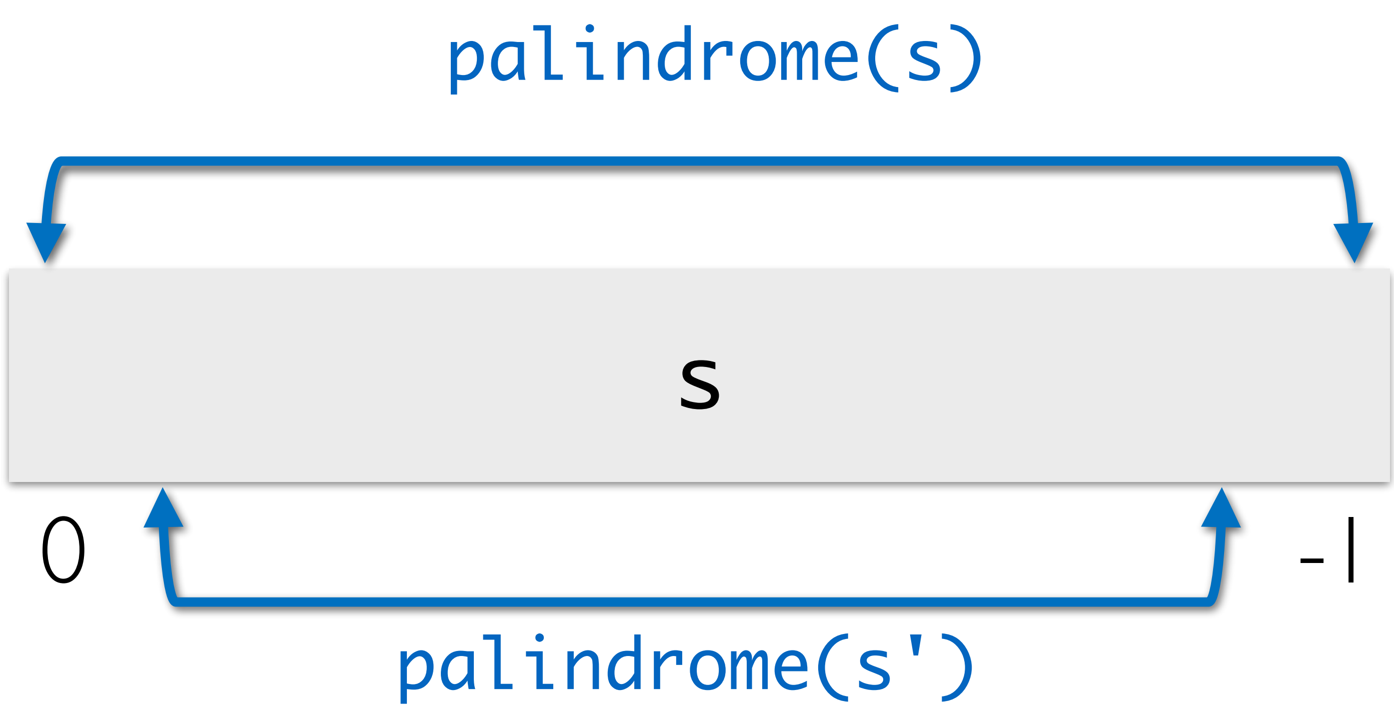
Recursive Approach

- DELEGATE the smaller problems to the recursion fairy *(formally known as induction hypothesis)* and assume they're solved correctly

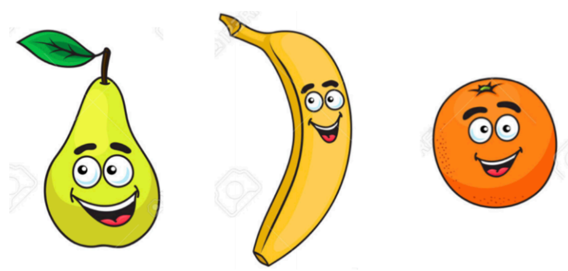


Recursive Approach

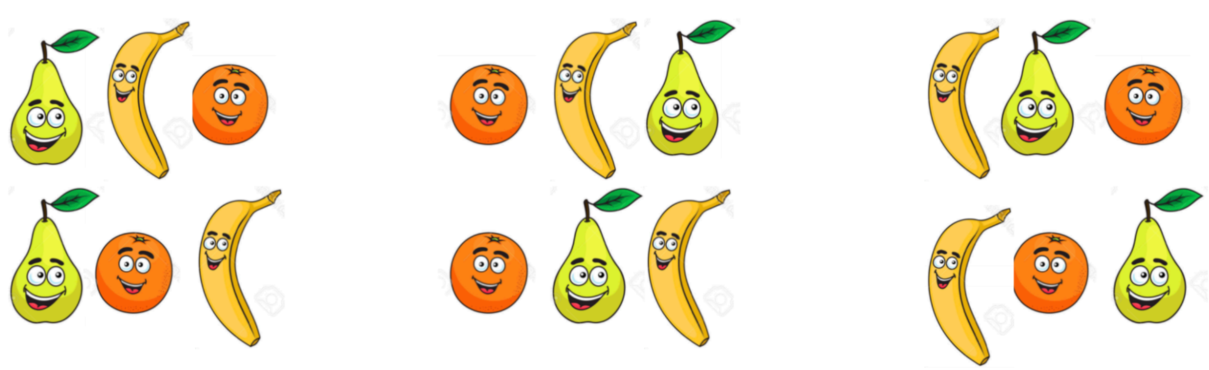
- COMBINE the solution(s) of the smaller subproblems to reach/return the solution
  - return `True` if `palindrome(s')` is `True` and `s[0]` is same as `s[-1]`



Factorial



How many ways can you arrange 3 items in a sequence?



3 items were arranged in 6 different ways. `Or 3x2x1`.

How about 4 items?

Factorial. Denoted `n!`

$n! = n * (n-1) * (n-2) * \ldots. * 2 * 1$

number of different the arrangements of n items.

factorial(n)

- $n! = n * (n - 1) * (n - 2) * \ldots * 2 * 1$
- $n! = n * (n - 1)!$
- Recursive case.

  `factorial(n)` is `n * factorial(n-1)`
- Base case.

  `factorial(0)` = 1

Summary

- Fruitful recursion: recursion that "computes and returns" values
- Remember to implement the base case!
- Remember to store the value returned by recursive calls!
- Debug using print statements

Recursion with Turtle Graphics

Turtle

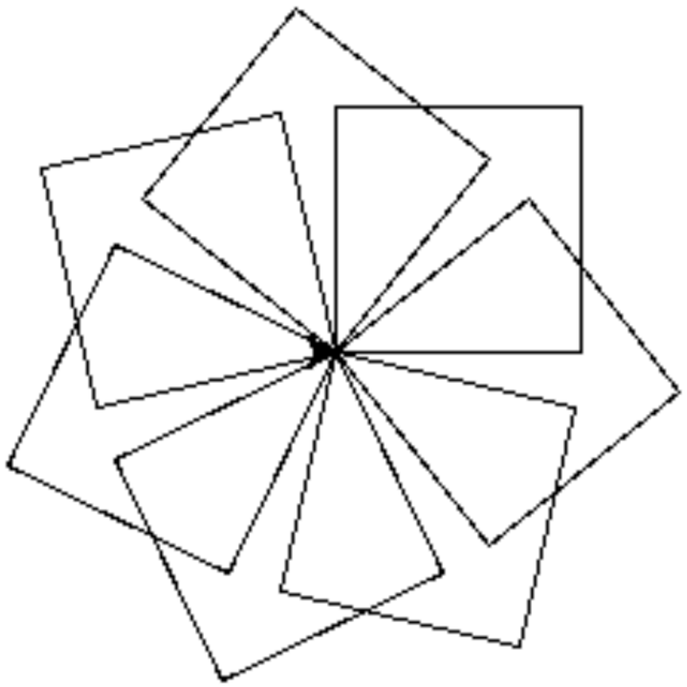
- Python has a built-in module named `turtle`.
  See the Python turtle module API for details.

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

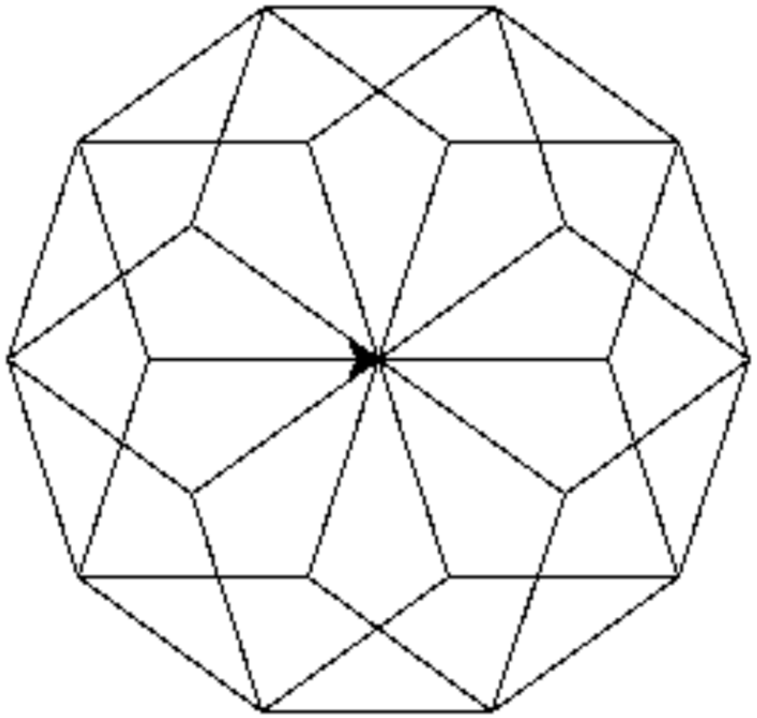
`fd(dist)`	turtle moves forward by *dist*
`bk(dist)`	turtle moves backward by *dist*
`lt(angle)`	turtle turns left *angle* degrees
`rt(angle)`	turtle turns right *angle* degrees
`pu()`	(pen up) turtle raises pen in belly
`pd()`	(pen down) turtle lower pen in belly
`pensize(width)`	sets the thickness 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*

Playing with Turtle: `polyFlow`

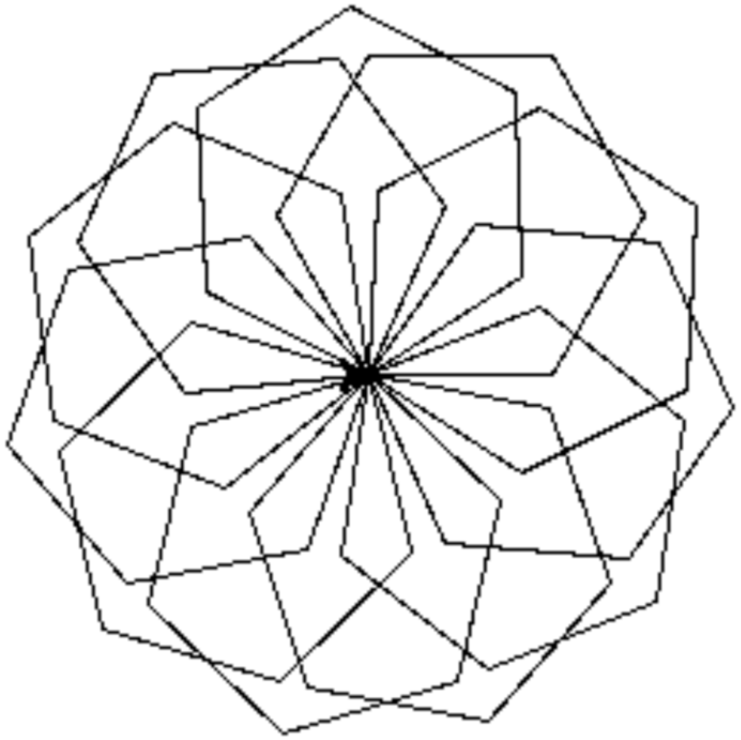
`polyFlow(7,4,80)`



`polyFlow(10,5,75)`

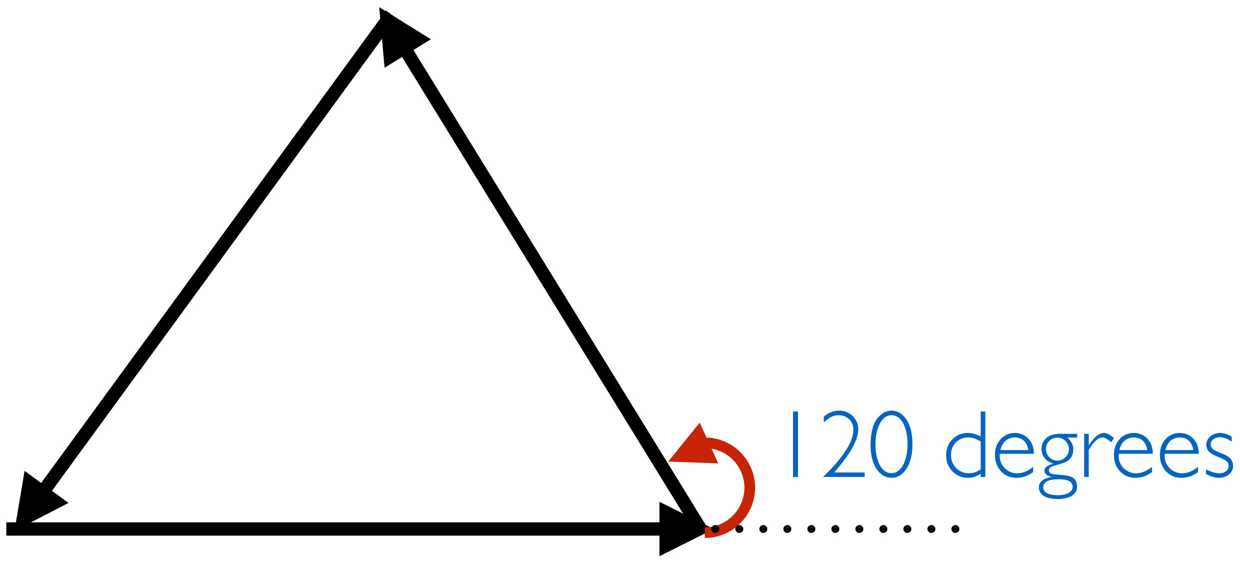


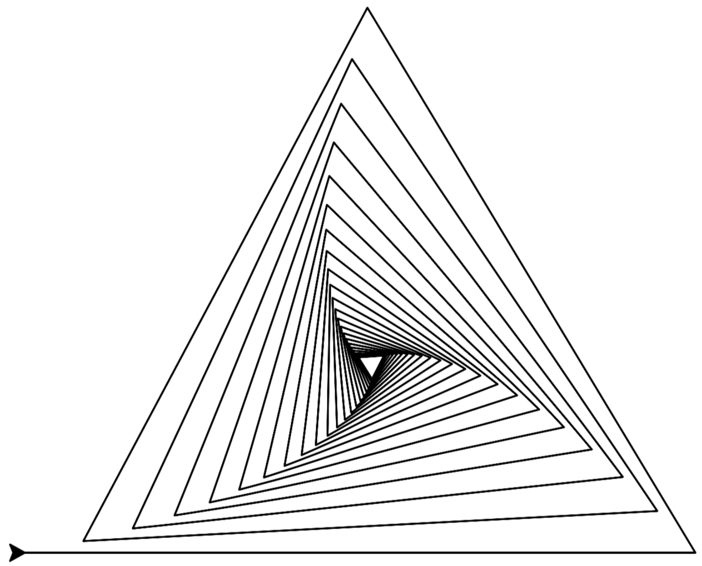
`polyFlow(11,6,60)`



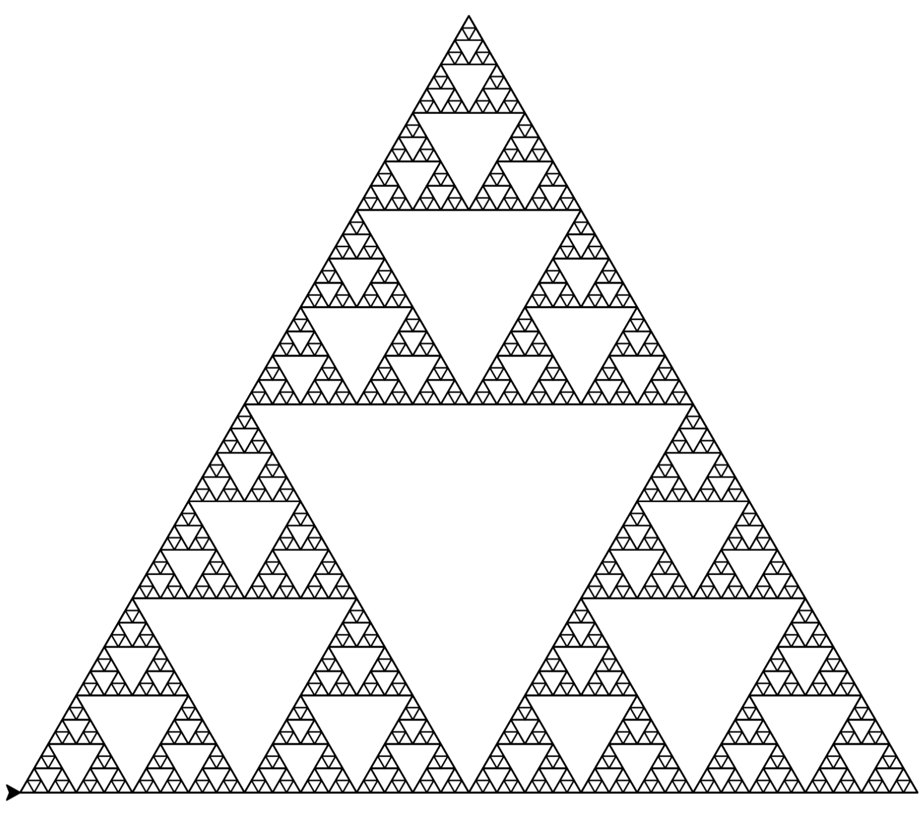
The Sun totally ruined by plans!
You can't see anything....

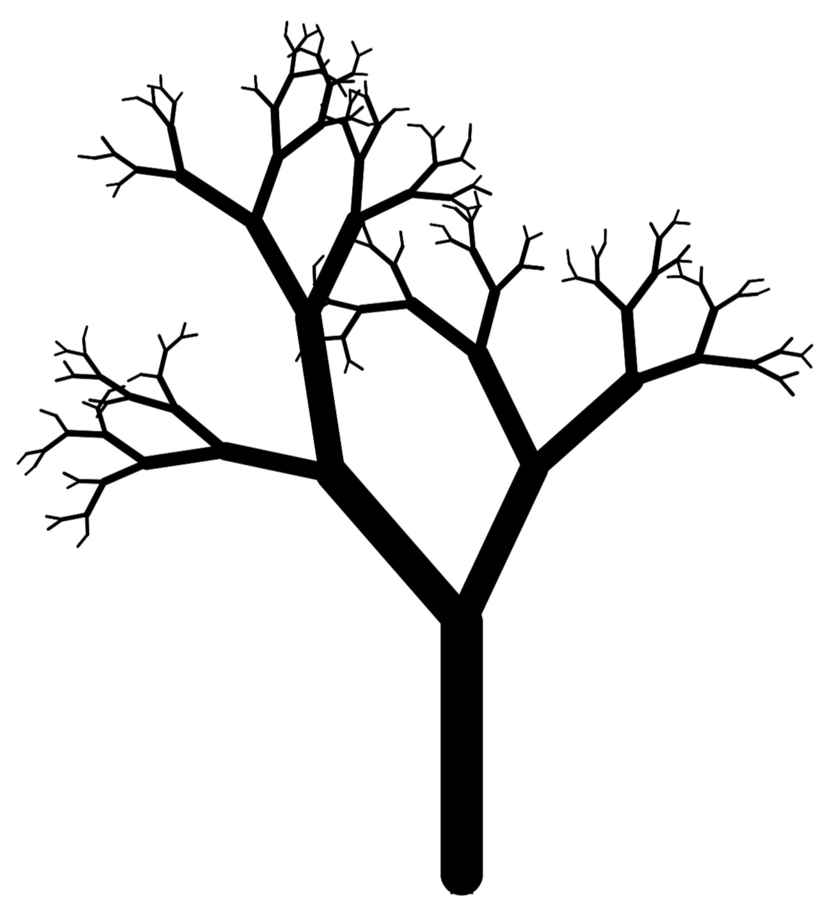
This is what I am drawing





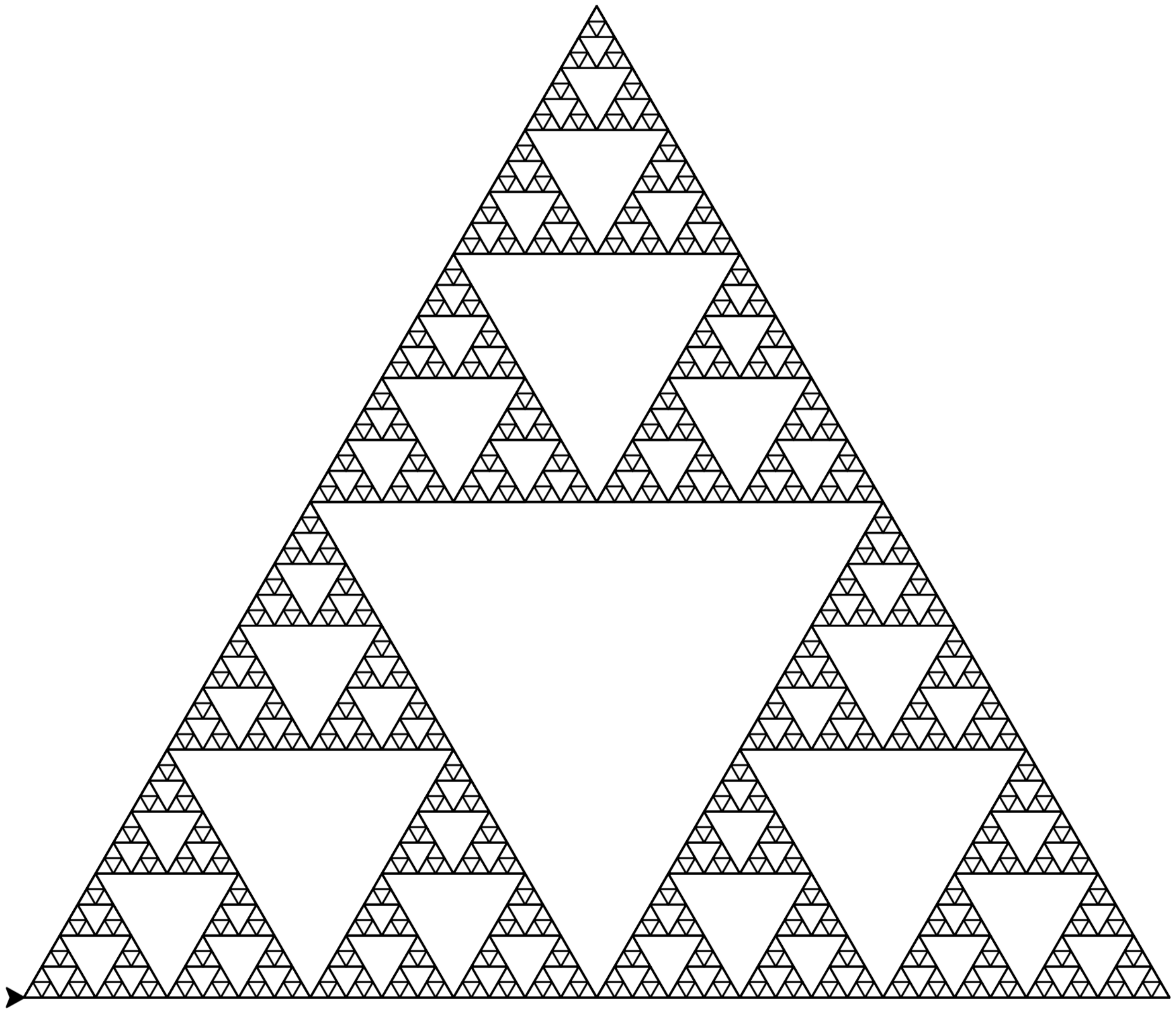
Graphical Recursion





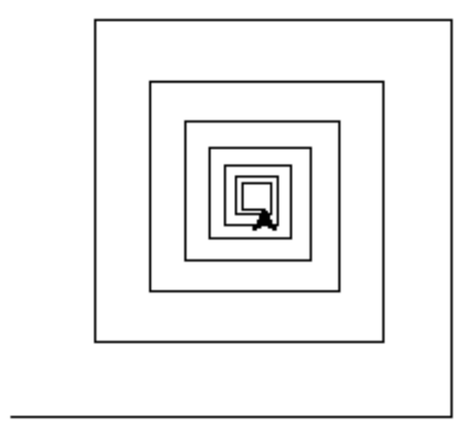
Overview

- Graphical recursion with a single recursive call
- Fruitful recursion with turtles
- Learn about **function invariance** in anticipation of multiple recursive calls

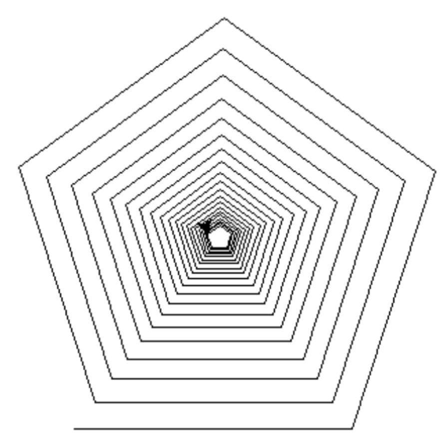


Sierpinski's Triangle

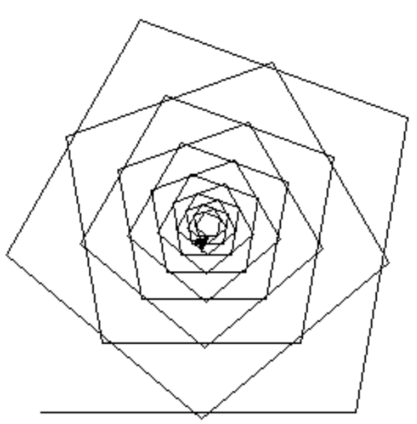
Single Recursive Call: Recursive Spirals



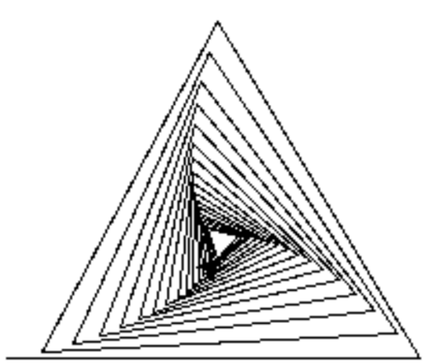
`spiral(200,90,0.9,10)`



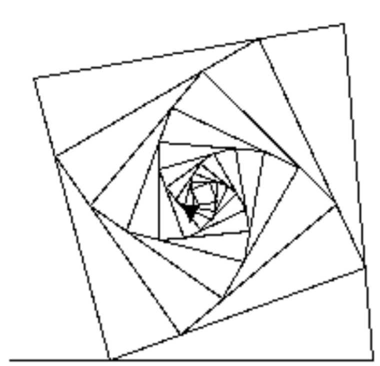
`spiral(200,72,0.97,10)`



`spiral(200,80,0.95,10)`

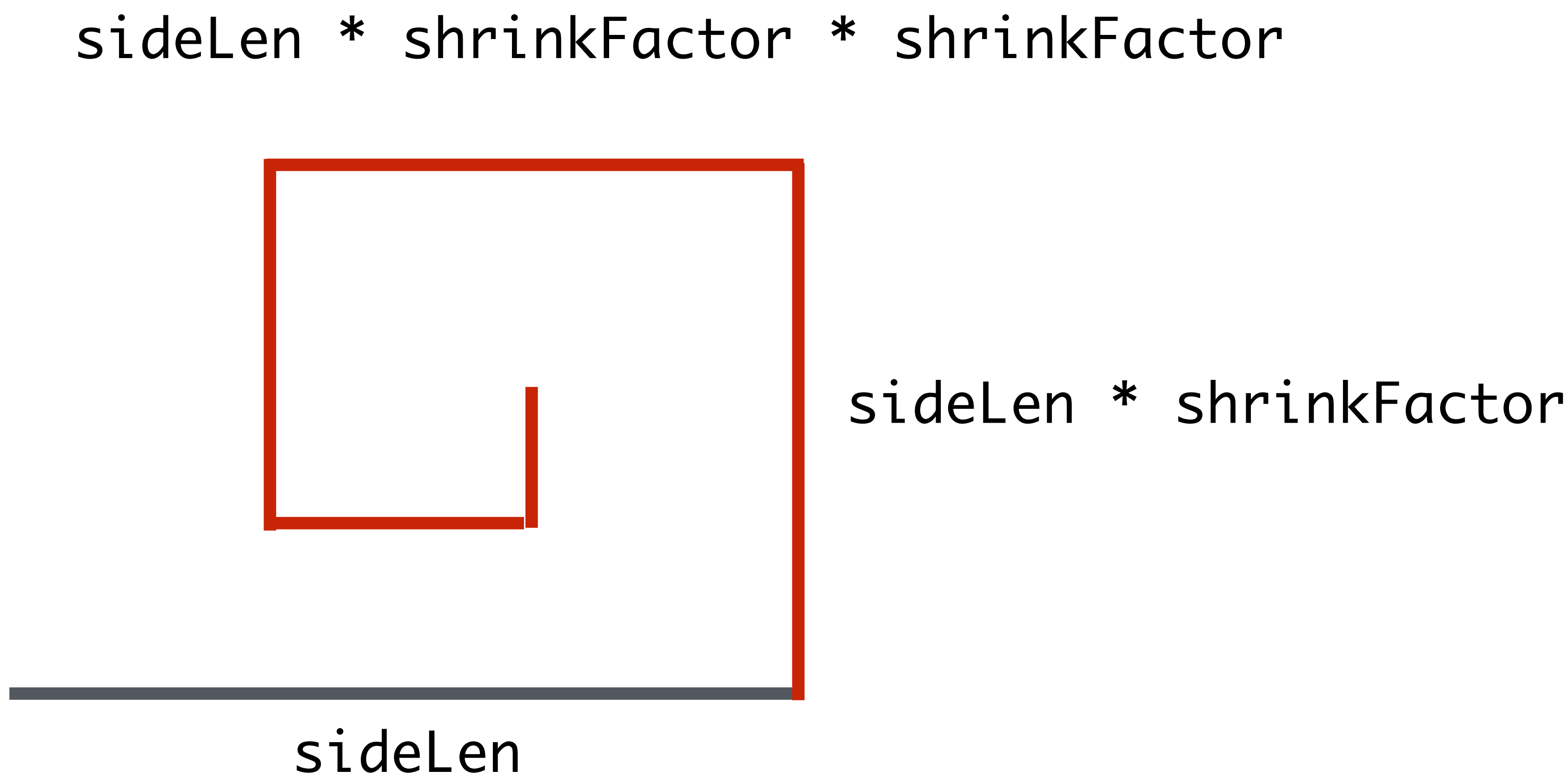


`spiral(200,121,0.95,15)`



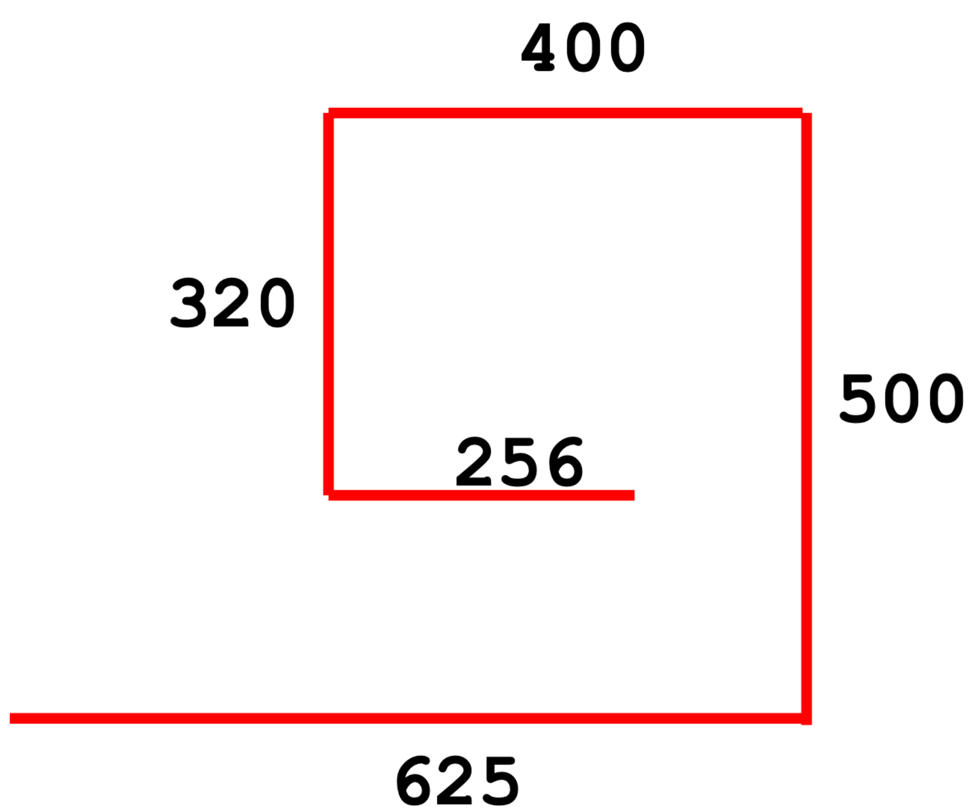
`spiral(200,95,0.93,10)`

Recursive Spirals



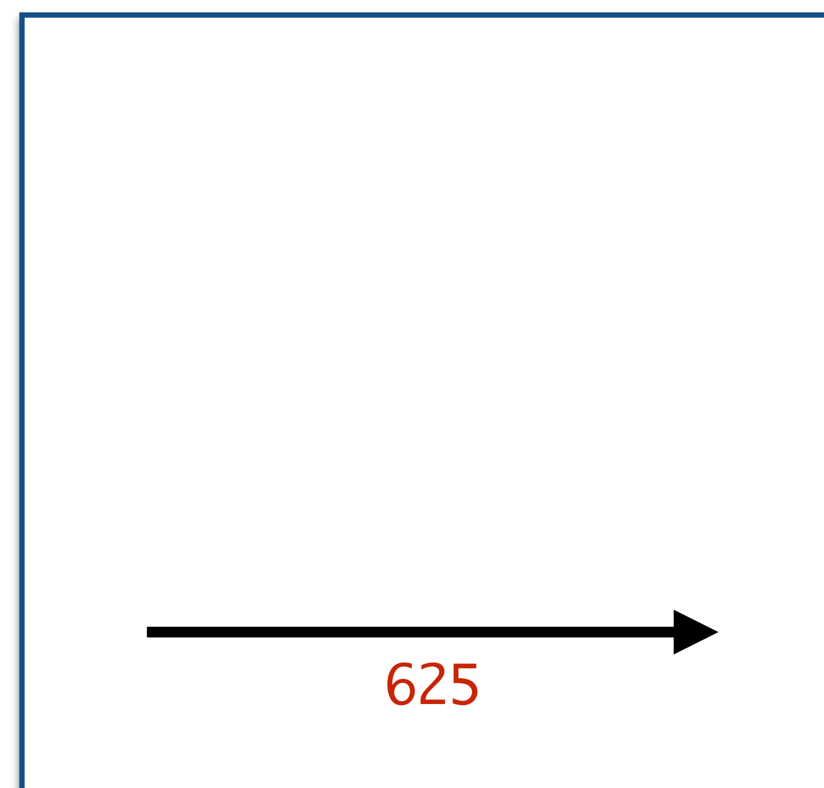
Function Frame Model to Understand `spiral`

```python
def spiral(sideLen, angle,
               scaleFactor, minLength):
    """Draw a spiral recursively."""

    if sideLen >= minLength:
        fd(sideLen)
        lt(angle)
        spiral(sideLen*scaleFactor,
               angle,
               scaleFactor,
               minLength)
```



spiral(625, 90, 0.8, 250)

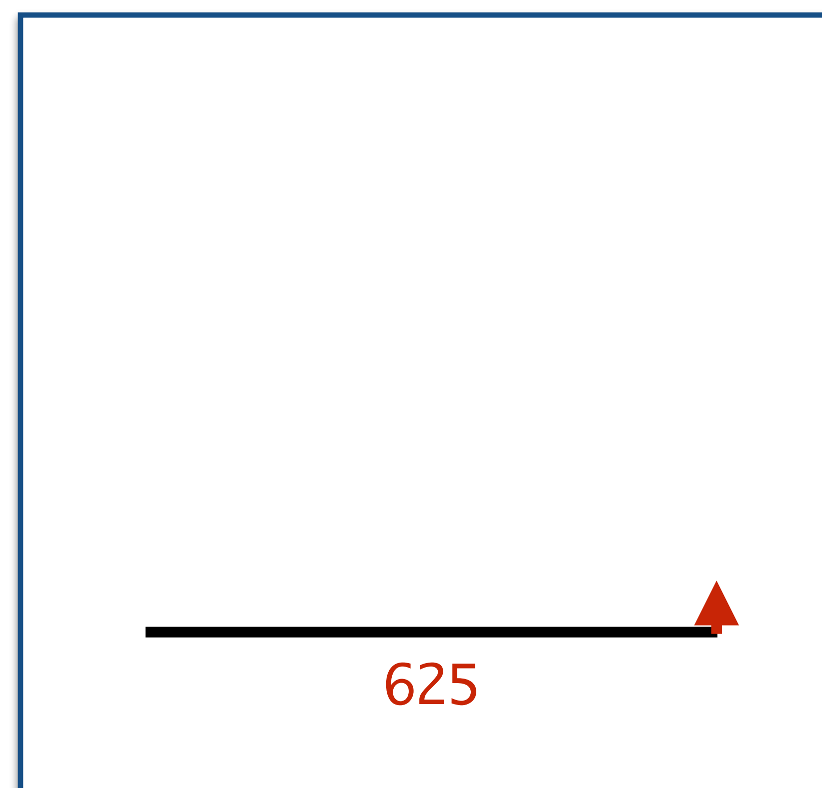
```
spiral(625, 90, 0.8, 250)

sideLen  625  .......

if sideLen > 250:
    fd(sideLen)
    lt(90)
    spiral(500, ...)
```



```
spiral(625, 90, 0.8, 250)
```

sideLen	625	·······

```
if sideLen > 250:
    fd(sideLen)
    lt(90)
    spiral(500, ...)
```

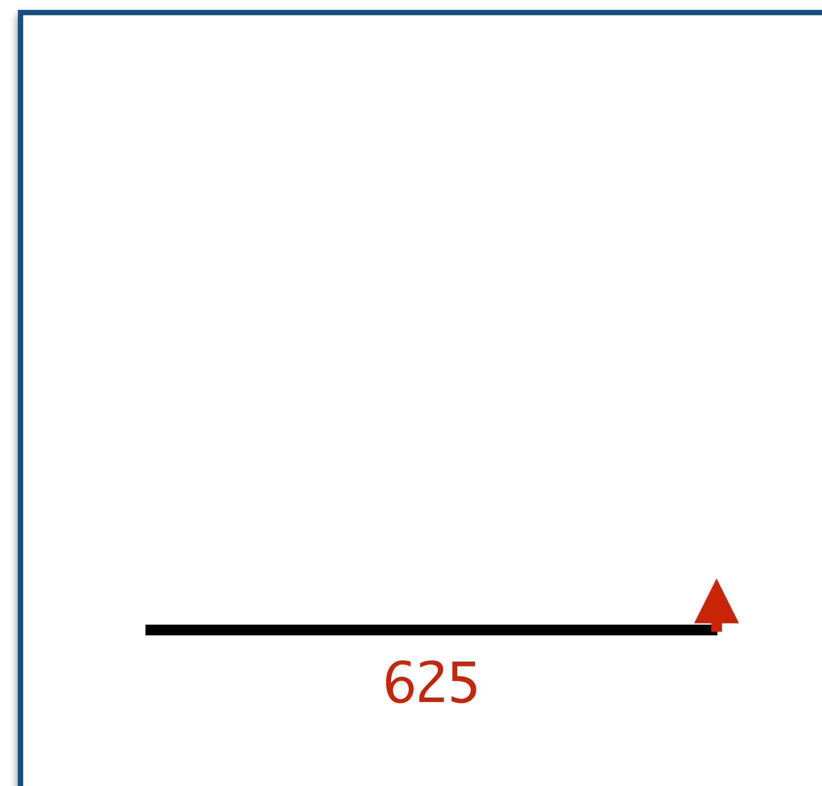


spiral(625, 90, 0.8, 250)

```
sideLen  625  .......

if sideLen > 250:
    fd(sideLen)
    lt(90)
    spiral(500, ...)
```

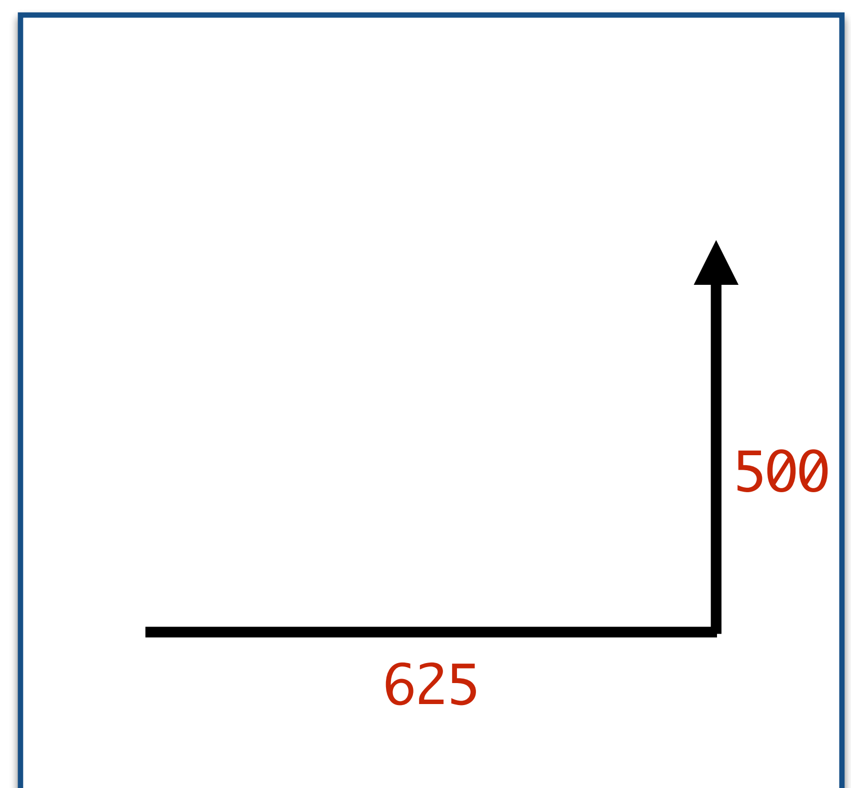
spiral(500, 90, 0.8, 250)

```
sideLen  500

if sideLen > 250:
    fd(sideLen)
    lt(90)
    spiral(400, ...)
```



```
spiral(625, 90, 0.8, 250)
sideLen  625  .......

if sideLen > 250:
    fd(sideLen)
    lt(90)
    spiral(500, ...)
```

```
spiral(500, 90, 0.8, 250)
sideLen  500

if sideLen > 250:
    fd(sideLen)
    lt(90)
    spiral(400, ...)
```

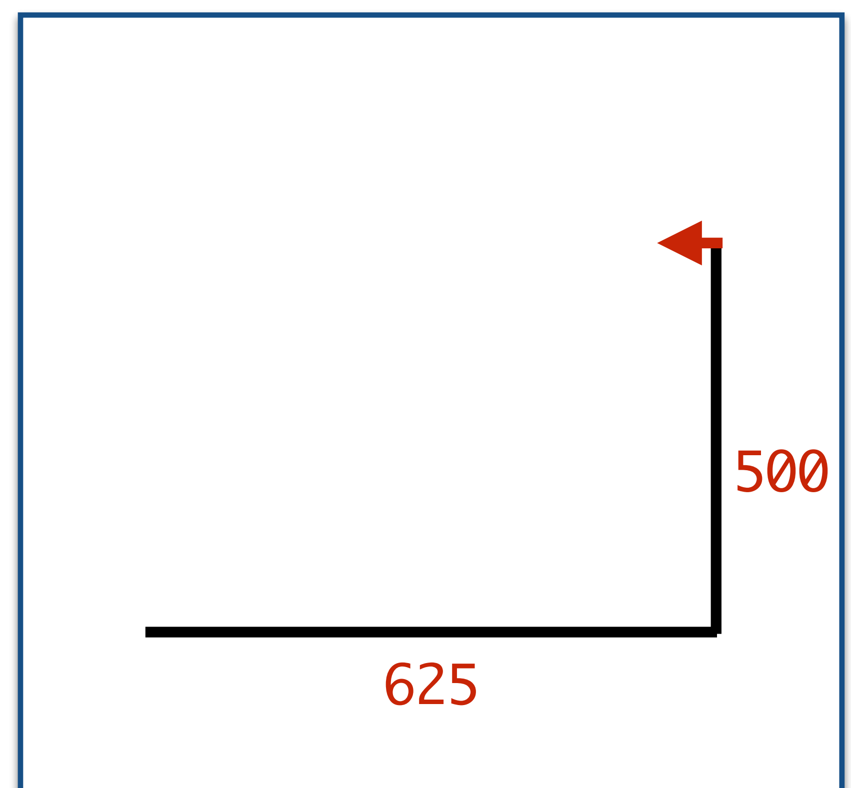


spiral(625, 90, 0.8, 250)

```
sideLen  625  .......

if sideLen > 250:
    fd(sideLen)
    lt(90)
    spiral(500, ...)
```

spiral(500, 90, 0.8, 250)

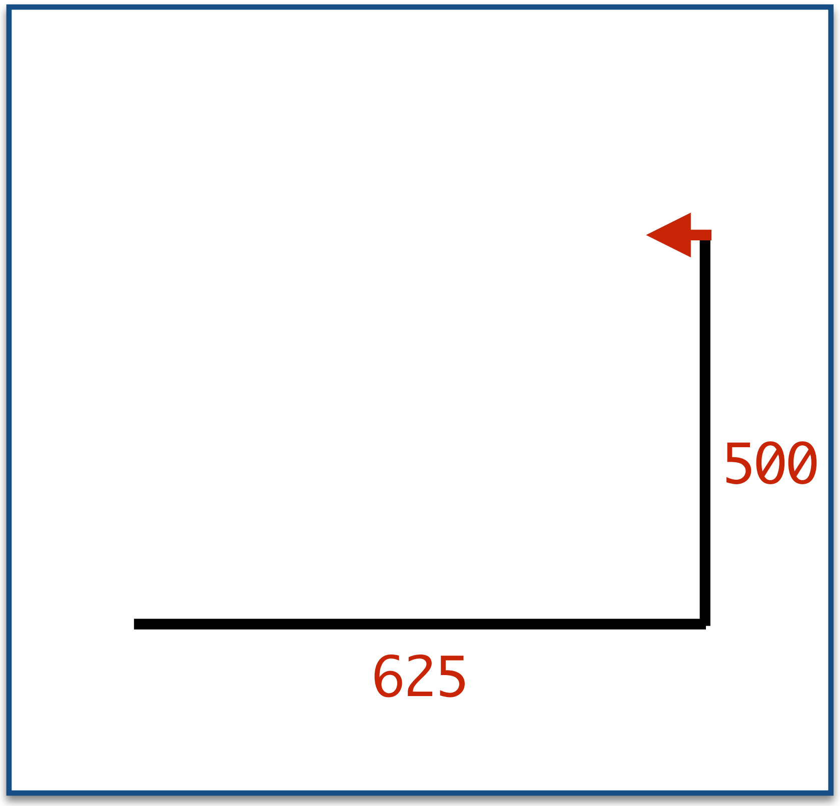
```
sideLen  500

if sideLen > 250:
    fd(sideLen)
--> lt(90)
    spiral(400, ...)
```



```
spiral(625, 90, 0.8, 250)
sideLen  625  .......

if sideLen > 250:
    fd(sideLen)
    lt(90)
    spiral(500, ...)
```

```
spiral(500, 90, 0.8, 250)
sideLen  500

if sideLen > 250:
    fd(sideLen)
    lt(90)
    spiral(400, ...)
```

```
spiral(400, 90, 0.8, 250)
sideLen  400

if sideLen > 250:
    fd(sideLen)
    lt(90)
    spiral(320, ...)
```



spiral(625, 90, 0.8, 250)

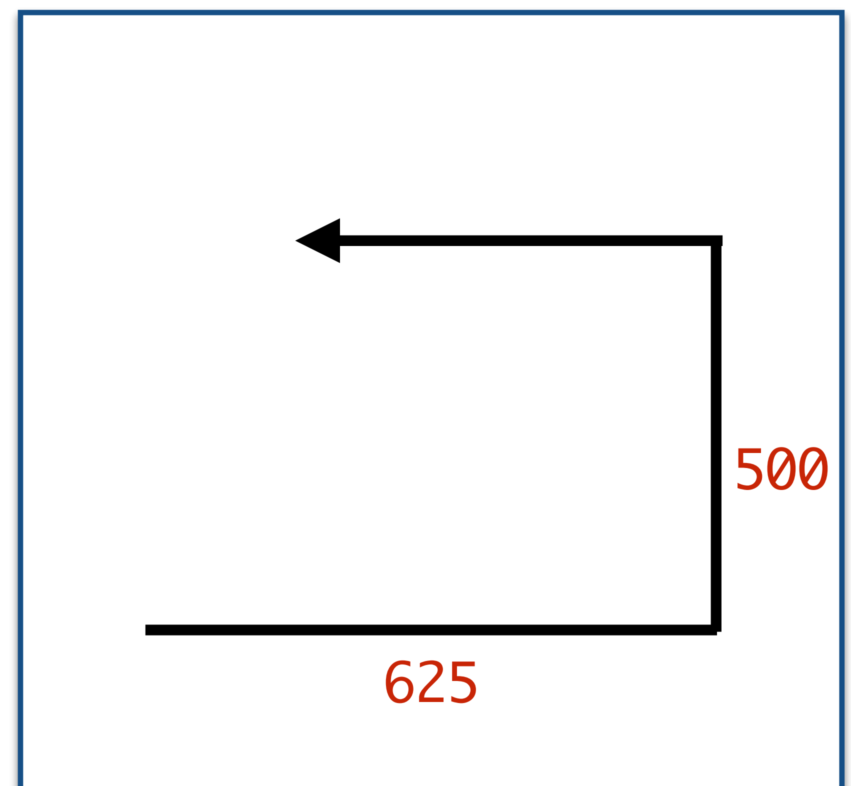
```
sideLen  625  .......

if sideLen > 250:
    fd(sideLen)
    lt(90)
    spiral(500, ...)
```

spiral(500, 90, 0.8, 250)

```
sideLen  500

if sideLen > 250:
    fd(sideLen)
    lt(90)
    spiral(400, ...)
```

spiral(400, 90, 0.8, 250)

```
sideLen  400

if sideLen > 250:
    fd(sideLen)
    lt(90)
    spiral(320, ...)
```



Invariant Spiralling

Invariance

- A function is invariant relative to an objects state if the state of the object is the same before and after a function is invoked

```
Do state change 1
Do state change 2
…
Do state change n-1
Do state change n
```

Perform changes to state

```
Recursive call to function
```

```
Undo state change n
Undo state change n-1
…
Undo state change 2
Undo state change 1
```

Undo state changes in opposite order

Fruitful Recursion with Turtles

See Lecture Jupyter Notebook

Acknowledgments

These slides have been adapted from:

- http://cs111.wellesley.edu/spring19 and
- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/