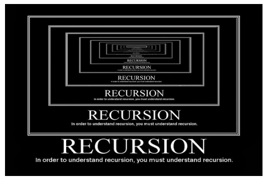
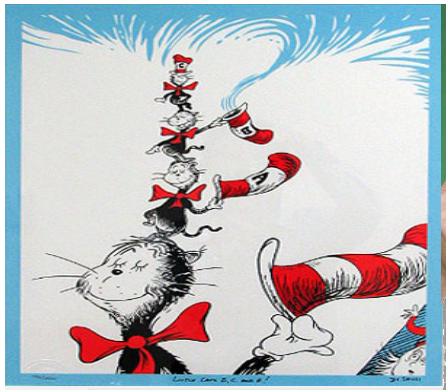
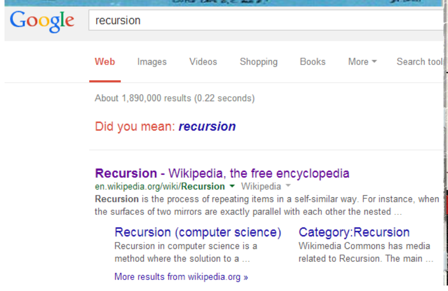
Introduction to Recursion


RECURSION
In order to understand recursion, you must understand recursion.

CS111 Computer Programming

Department of Computer Science
Wellesley College





About 1,890,000 results (0.22 seconds)

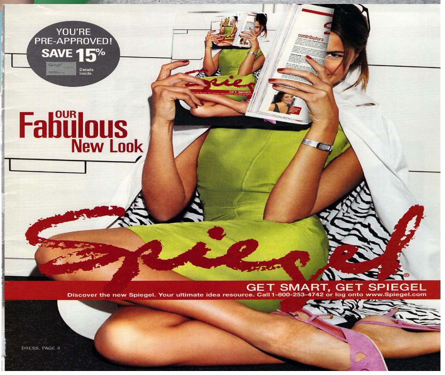
Did you mean: ***recursion***

Recursion - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Recursion ▾ Wikipedia ▾
Recursion is the process of repeating items in a self-similar way. For instance, when the surfaces of two mirrors are exactly parallel with each other the nested ...

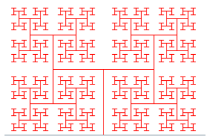
Recursion (computer science)
Recursion in computer science is a method where the solution to a ...

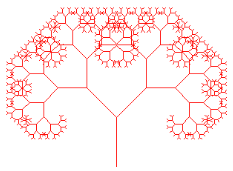
Category:Recursion
Wikimedia Commons has media related to Recursion. The main ...

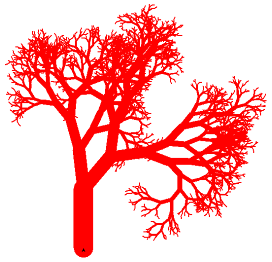
More results from wikipedia.org »

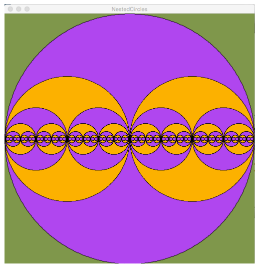


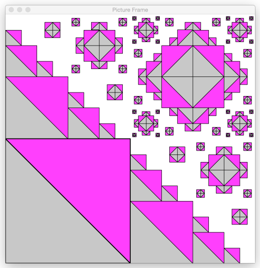
Recursive Patterns

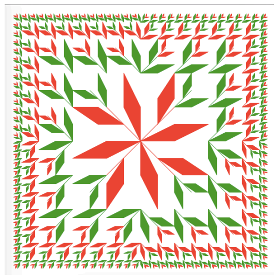












What is Recursion?

Recursion is an instance of the **divide/conquer/glue** problem solving strategy in which one or more subproblems is a smaller instance of the original problem.

A recursive function is a function that invokes itself.

Designing recursive functions requires:

- Understanding the difference between defining a function and calling a function
- Understanding that it's OK for a function to call another function that hasn't yet been defined (or is in the process of being defined).
- Thoughtfully decomposing a problem into subproblems

Review: functions calling other functions

Which of the following work? Which don't work? Why?

```
def print2(s):
    print s
    print s

def print4(s):
    print2(s)
    print2(s)

print4('foo')
```

```
def print4(s):
    print2(s)
    print2(s)

def print2(s):
    print s
    print s

print4('foo')
```

```
def print4(s):
    print2(s)
    print2(s)

print4('foo')

def print2(s):
    print s
    print s
```

countDown

Suppose we want to write a function that prints the integers from **n** down to 1 (**without using loops**):

```
def countDown(n):
    '''Prints integers from n down to 1'''
    if n < 1:
        pass # Do nothing
    else:
        print n
        countDown(n-1)
```

```
In [3]: countDown(5)
5
4
3
2
1
```

Structure of Recursion

All recursive functions have two parts:

- **A base case**: a simple case where the result is so simple, it can just be returned. In this case the function does not invoke itself, since there is no need to decompose the problem into subproblems.
- **a recursive case**: a case where the problem is decomposed into subproblems and at least one of the subproblems is solved by invoking the function being defined, i.e., the function is invoked in its own body.

countDown: Base Case

The base case. When is the problem so simple that we can solve it trivially and we needn't decompose it into subproblems.

```
def countDown(n):
    '''Prints integers from n down to 1'''
    if n < 1:
        pass #Do nothing
```

countDown: Recursive Case

The recursive case. For all instances of the problem not covered by the base case, we'll decompose the problem into subproblems, at least one of which is a smaller instance of the **countDown** problem and can be solved by invoking the **countDown** function.

```
def countDown(n):
    '''Prints integers from n down to 1'''

    if n < 1:
        pass # Do nothing

    else:
        print n
        countDown(n-1)
```

tower

Suppose we want to write a function that prints a tower based on the characters of the input string **name** starting with len(name) characters down to the last character :

it's your turn

```
def tower(name):
'''Prints a tower based on the string name from
len(name) characters down to the last character
'''
```

```
In [6]:
tower('Wellesley')
Wellesley
ellesley
llesley
lesley
esley
sley
ley
ey
y
```

What does this function do?

it's your turn

```
def mystery(n):
    if n < 1:
        pass
    else:
        mystery(n - 1)
        print n
```

countDownUp

Suppose we want to write a function that prints the integers from **n** down to 1 and then from 1 up to **n**:

```
def countDownUp(n):
    '''Prints integers from n down
       to 1 and then from 1 up to n'''
```

```
In [6]: countDownUp(4)
4
3
2
1
1
2
3
4
```

sumUpTo

Suppose we want to write a function that returns the sum of all the positive integer numbers up to n :

```
def sumUpTo(n):
'''write a function that returns the sum of all
positive integer numbers up to n'''
```

```
In [6]:sumUpTo(6)
Out [6]:21

In [6]:sumUpTo(10)
Out [6]:55
```

factorial

Suppose we want to write a function that returns n!:

n! = n * (n-1) * (n-2) * ...* 1

```
def factorial(n):
'''write a function that returns n!'''
```

it's your turn

```
In [6]:factorial(3)
Out [6]:6

In [7]:factorial(5)
Out [7]:120
```

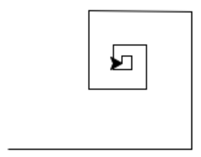
Turtle Graphics

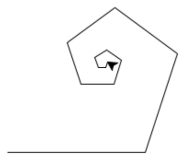
Python has a built-in module named turtle. See the Python turtle module API for details.

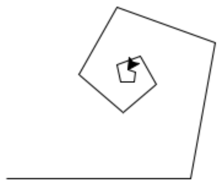
Use `from turtle import *` to use these commands:

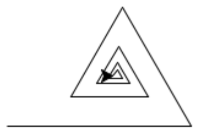
`fd(dist)`	turtle moves forward by *dist*
`bk(dist)`	turtle moves backward by *dist*
`lt(angle)`	turtle turns left *angle* degrees
`rt(angle)`	turtle turns right *angle* degrees
`pu()`	(pen up) turtle raises pen in belly
`pd()`	(pen down) turtle lower pen in belly
`pensize(width)`	sets the thickness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

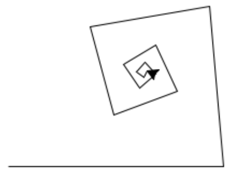
Spiraling Turtles: A Recursion Example


`spiral(200,90)`


`spiral(150,72)`


`spiral(150,80)`


`spiral(150,120)`


`spiral(200,95)`

spiral(sideLen,angle)

- **sideLen** is the length of the current side
- **angle** is the amount the turtle turns left to draw the next side

```
def spiral(sideLen, angle):
 """Keeps drawing lines, each line is 75% the
 length of the previous line, and the
 recursion stops when the length of the line is
 less than 5"""
```

spiral-Base case

```
def spiral(sideLen, angle):
 """Keeps drawing lines, each line is 75% the
 length of the previous line, and the
 recursion stops when the length of the line is
 less than 5"""

  if sideLen < 5:
      pass
```

spiral-Recursive case

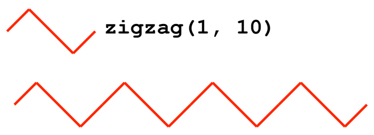
```
def spiral(sideLen, angle):
 """Keeps drawing lines, each line is 75% the
 length of the previous line, and the
 recursion stops when the length of the line is
 less than 5"""

  if sideLen < 5:
      pass
  else:
```

Invariant Spiraling

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

```
# Draws a spiral. The state of the turtle (position,
# color, heading, etc.) after drawing the spiral is the
# same as before drawing the spiral.
def spiralBack(sideLen, angle):
    if sideLen < 5:
        pass
    else:
        fd(sideLen)
        lt(angle)
        spiralBack(sideLen*0.75, angle)
        rt(angle)
        bk(sideLen)
```

zigzag

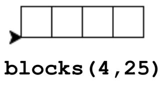


```
# Draws the specified number of zigzags with the specified
# length.
def zigzag(num, length):
      if num <= 0:
         pass
      else:
           lt(45)
           fd(length)
           rt(90)
           fd(2*length)
           lt(90)
           fd(length)
           rt(45)
           zigzag(num-1, length)
```

it's your turn

Exercise 9: modify zigzag to make the turtle's state invariant.

blocks and blockHelper



Suppose we want to write a recursive function that draws a sequence of n blocks

```
# Draws the specified number of blocks with the specified
# length. The state of the turtle (position,
# color, heading, etc.) after drawing the blocks is the
# same as before drawing the blocks.
def blocks(num, length):
```

it's your turn

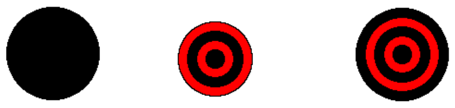
We can use a helper function that draws a single block:

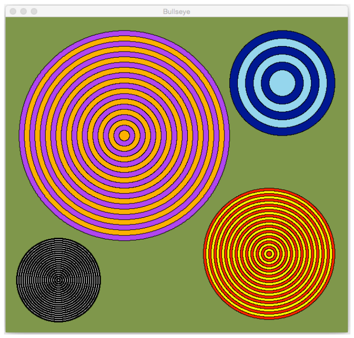
```
# Draws a square using recursion
# with the specified side length.
def blockHelper(numSides, length):
```

drawTarget: recursive function

```
def drawTarget(canvas,x,y,radius,thickness,color1,color2):
    '''On the specified canvas, draws a bullseye target with
    the given radius, centered at (x,y) with alternating colors,
    color1 and color2, where color1 is the outermost color;
    thickness is the width of each "band" in the ring;
    thickness is also the minimum radius of a drawn circle'''
```

Hint: how can we decompose the problem into two subproblems such that one of the subproblems involves drawing a target?





What about this pattern?