Introduction to Recursion

Reminder: DCG

CS111 Computer Programming
Department of Computer Science
Wellesley College

What is Recursion?

Divide-conquer-glue...
...where the sub-problems involve the problem itself!

With recursion, the solution to a problem depends on solutions to smaller instances of the same problem

A recursive function is a function that invokes itself.
Review: functions calling other functions
(in anticipation of writing functions that call themselves)

Which would work? Why/why not?

def print2(s):
    print s
    print s
def print4(s):
    print2(s)
    print2(s)
def print2(s):
    print s
    print s
print4('okay')
def print2(s):
    print s
    print s
print4('okay')
def print4(s):
    print2(s)
    print2(s)
print4('okay')
def print2(s):
    print s
    print s

Our first recursive function: countDown
Let's 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)

Decomposition
We can think of the countDown(n) as composed of the print(n) statement and the invocation of countDown(n-1). And so on.

Concepts in this slide:
- In recursion, the smaller problems are the same as the problem we're trying to solve.
- The recursive case makes the problem smaller. The base case can be omitted.

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

Knowing when to stop
At some point, there is no more decomposition, we have come to a point where we'll need to stop.
This is similar to loops. The base case is what tells the recursion to stop.

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)

def countDownImplicitBase(n):
    '''Prints integers from n down to 1'''
    if n > 0:
        print n
countDownImplicitBase(n-1)

To notice:
The recursive step does two things:
a) performs an action that contributes to the solution
b) Invokes the function by making the problem smaller

It is possible to omit the base case when no action is happening in it.
**Structure of Recursion**

All recursive functions must have two types of cases:

- **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.

- **RECURSIVE case**: a case where the problem is decomposed into subproblems
  - at least one of the subproblems is solved by invoking the function being defined, i.e., the function is invoked in its own body. You should assume the recursive function works correctly for the smaller subproblems (this is known as “wishful thinking”)

---

**Invocation of `countDown(3)`**

**Anatomy of function call frames**

```
def countDown(n):
    '''Prints integers from n down to 1'''
    if n>0:
        print(n)
        countDown(n-1)
```

- **Control arrow shows what's currently being evaluated in function body**
- **Function call frame**
- **Local variable in the frame for:**
  1. Each parameter
  2. Each local name

---

**Review: function call frames**

Recursion isn’t magic. It works because of the frame model for functions we introduced back in Lecture 04.

Recursion I 18-9

---

**Invocation of `countDown(3)`**

Draw the diagram of function call frames

```
def countDown(n):
    '''Prints integers from n down to 1'''
    if n>0:
        print(n)
        countDown(n-1)
```

- **There is a local variable in the frame for:**
  1. Each parameter
  2. Each local name

---

**Recursion I 18-10**
Recursion **GOTCHA! #1:** subproblem not smaller

The problem that you are solving recursively must get smaller each time you recur, i.e., you must get closer to the base case. Otherwise, the recursion will not terminate -- a so-called infinite recursion.

```python
def countDown(n):
    if n < 1:  # Base case
        pass  # Do nothing
    else:     # Recursive case
        print n
        countDown(n)
```

Recursion **GOTCHA! #2:** missing base case

The recursion must eventually reach a base case in order to end. If it doesn't, that's another way to get an infinite recursion.

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

In practice, the infinite recursion examples will terminate when Python runs out of resources for creating function call frames, leading to a maximum recursion depth exceeded error message:

```
In [2]: countDown(3)
3
2
1
0
-1
-2
-3
...
RuntimeError: maximum recursion depth exceeded while calling a Python object foo
```

**What does this function do?**

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

What does mystery(3) print?

**countDownUp**

Let's write a function that prints the integers from n down to 1 and then from 1 up to n:

```python
def countDownUp(n):
    '''Prints integers from n down to 1 and then from 1 up to n
    '''
    In [1]: countDownUp(4)
    4
    3
    2
    1
    1
    2
    3
    4
```
countDownUp - Base Case

When is the problem so simple that we needn't decompose it into subproblems? What code do we want to execute in this case?

```python
def countDownUp(n):
    '''Prints integers from n down to 1 and then from 1 up to n.'''
    if n < 1:
        pass  # Do nothing
    else:
        In []: countDownUp(4)
        4
        3
        2
        1
        1
        2
        3
        4
```

countDownUp - Recursive Case

How can we decompose the problem into subproblems so that one of the subproblems can be solved using `countDownUp`?

```python
def countDownUp(n):
    '''Prints integers from n down to 1 and then from 1 up to n.'''
    if n < 1:
        pass  # Do nothing
    else:
        In []: countDownUp(4)
        4
        3
        2
        1
        1
        2
        3
        4
```

drawTarget: base case?

```python
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.'''
```

drawTarget: recursive case?

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

```python
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.'''
```

Target Practice (concentric circles)

Let's draw PS04 `cs1graphics` “targets” using recursion:

```python
def drawDisc(canvas, x, y, r, color):
    '''Draw a circle centered at point (x,y) on the canvas with given radius r and color.'''
    c = Circle(radius, Point(x, y))
    c.setFillColor(color)
    canvas.add(c)

drawDisc invocations

paper = Canvas(400, 300, 'darkolivegreen4', 'drawDisc')
drawDisc(paper, 100, 175, 75, 'blue')
drawDisc(paper, 250, 75, 50, 'red')
drawDisc(paper, 275, 225, 35, 'yellow')
```

Define drawNestedCircles

```python
def drawNestedCircles(canvas, x, y, radius, minRadius, color1, color2):
    '''On the specified canvas, draws a nested circle pattern. The largest circle is centered at (x,y) with the given radius and is filled with color1. There are two nested subpatterns, each with half the radius, internally tangent along the main horizontal diameter, with colors alternating between color2 and color1. No circle whose radius is less than minRadius is drawn.'''
```

```python
def drawNestedCircles(canvas, x, y, radius, minRadius, color1, color2):
    '''On the specified canvas, draws a nested circle pattern. The largest circle is centered at (x,y) with the given radius and is filled with color1. There are two nested subpatterns, each with half the radius, internally tangent along the main horizontal diameter, with colors alternating between color2 and color1. No circle whose radius is less than minRadius is drawn.'''
```

Extra: it's your turn

```python
def drawNestedCircles(canvas, x, y, radius, minRadius, color1, color2):
    '''On the specified canvas, draws a nested circle pattern. The largest circle is centered at (x,y) with the given radius and is filled with color1. There are two nested subpatterns, each with half the radius, internally tangent along the main horizontal diameter, with colors alternating between color2 and color1. No circle whose radius is less than minRadius is drawn.'''
```

DrawNestedCircles invocations

```python
drawNestedCircles(paper, 300, 300, 300, 3, 'midnightblue', 'palegreen')
```