Recall big idea #3:
Divide, conquer & glue (DCG)

Divide
problem P into subproblems.

Conquer
each of the subproblems, &

Glue (combine)
the solutions to the subproblems into a solution S for P.

DCG to make cool pictures

Today we’ll see how to use DCG to make complex and interesting pictures in a simple way.

But first it will help to define a more abstract notion of “picture” starting with cs1graphics.
Peter Henderson’s Picture Language

Picture abstraction on top of cs1graphics
- Inspired by Peter Henderson’s picture language.
- We define a “picture” as a cs1graphics Drawable object that’s 200x200 centered at reference point (0,0).
- By fixing the size of a picture, we never have to worry about pictures of different sizes.
- Rotations and flips of pictures by angles that are multiples of 90 degrees result in another picture!
- We will define many functions that take picture(s) as inputs and return a picture as an output.

Some Primitive Pictures
- **bp** (blue patch)
- **rp** (red patch)
- **gw** (green wedge)

Sample primitives defined in picture.py [1]
```python
def patch(color):
    pic = Square(200)  # centered by default
    pic.setFillColor(color)
    pic.setBorderColor('black')
    return pic

rp = patch('red')

def wedge(color):
    pic = Polygon(Point(100, 0),
                   Point(100,100),
                   Point(-100, 100))
    # Shift reference point from (100,0) to (0,0)
    pic.adjustReference(-100,0)
    pic.setFillColor(color)
    pic.setBorderColor(color)  # no border!
    return pic

gw = wedge('green')
```
Sample primitives defined in picture.py [2]

```python
def checkmark(downColor, upColor):
    pic = Layer()
    downstroke = Path(Point(-100,0), Point(0,100))
    downstroke.setBorderColor(downColor)
    pic.add(downstroke)
    upstroke = Path(Point(0,100), Point(100,-100))
    upstroke.setBorderColor(upColor)
    pic.add(upstroke)
    return pic

mark = checkmark('red', 'blue')
```

```
def empty():
    return Layer()
emptyPic = empty()
```

Displaying Pictures

```python
def displayPic(pics):
    '''Display picture in 600x600 canvas'''
    frame = Canvas(600, 600,'white', 'Picture Frame')
    # Clone pic before changing it; otherwise
    # it wouldn't be a picture anymore!
    framedPic = pic.clone()
    framedPic.scale(3) # scale by 3 to fill 600x600 canvas
    framedPic.moveTo(300, 300) # move to center of canvas
    frame.add(framedPic)
```

* The definition of displayPic is a tad more complex than shown here to allow
for a closeAllPics() function that closes all canvases created by displayPic.

Clockwise rotations of pictures

```python
clockwise90(p); # Returns new picture that's p rotated 90° clockwise
```

```python
clockwise180(p); # Returns new picture that's p rotated 180° clockwise
```

```python
clockwise270(p); # Returns new picture that's p rotated 270° clockwise
```

Defining picture rotations in picture.py

```python
def clockwisePic(pic, angle):
    newPic = pic.clone() # create new pic by cloning it.
    newPic.rotate(angle) # if angle is a multiple of 90,
    # result still satisfies
    # definition of picture.
    return newPic
```

```python
def clockwise90(pic):
    return clockwisePic (pic, 90)
```

```python
def clockwise180(pic):
    return clockwisePic (pic, 180)
```

```python
def clockwise270(pic):
    return clockwisePic (pic, 270)
```

* To notice:
  • clockwisePic returns a new picture.
  • The three other functions invoke the function clockwisePic with a
    hard-coded angle value.
Flipping pictures

- `flipAcrossVert(p);` # Returns new picture that’s p flipped across vertical axis
- `flipAcrossHoriz(p);` # Returns new picture that’s p flipped across horizontal axis
- `flipAcrossDiag(p);` # Returns new picture that’s p flipped across 45-degree axis

```
def flipPic(pic, angle):
    newPic = pic.clone() # create new pic by cloning it.
    newPic.flip(angle) # if angle is a multiple of 45,
                      # result still satisfies
                      # definition of picture.
    return newPic
```

```
def flipAcrossVert(pic):
    return flipPic(pic, 0)

def flipAcrossHoriz(pic):
    return flipPic(pic, 90)

def flipAcrossDiag(pic):
    return flipPic(pic, 45)
```

Concepts in this slide:
Making use of abstraction for creating new operations.

Defining picture flipping in picture.py

```
def flipPic (pic, angle):
    newPic = pic.clone() # create new pic by cloning it.
    newPic.flip(angle) # if angle is a multiple of 45,
                      # result still satisfies
                      # definition of picture.
    return newPic
```

To notice:
- `flipPic` returns a new picture.
- The three other functions invoke the function `flipPic` with a hard-coded angle value.

Overlaying pictures

```
def overlay(pic1, pic2):
    '''Returns a new pic in which pic1 appears on top of pic2'''
    newPic = Layer()
    newPic.add(pic2) # bottom pic goes first
    newPic.add(pic1) # top pic goes last
    return newPic
```

```
def overlay(mark,gl):
    overlay(gl,mark)
    overlay(gl,mark)
```

Concepts in this slide:
new operation for pictures: overlay

```
def fourPics(a, b, c, d):
    newPic = Layer()
    aHalf = a.clone() # aHalf = a.clone()
    aHalf.scale(0.5) # aHalf.scale(0.5)
    bHalf = b.clone() # bHalf = b.clone()
    bHalf.scale(0.5) # bHalf.scale(0.5)
    cHalf = c.clone() # cHalf = c.clone()
    cHalf.scale(0.5) # cHalf.scale(0.5)
    dHalf = d.clone() # dHalf = d.clone()
    dHalf.scale(0.5) # dHalf.scale(0.5)
    aHalf.move(-50,-50) # aHalf.move(-50,-50)
    bHalf.move(50,50) # bHalf.move(50,50)
    cHalf.move(-50,50) # cHalf.move(-50,50)
    dHalf.move(50,50) # dHalf.move(50,50)
    newPic.add(aHalf)
    newPic.add(bHalf)
    newPic.add(cHalf)
    newPic.add(dHalf)
    return newPic
```

Concepts in this slide:
a combinator operation: `fourPics` that creates a new picture.

```
def fourPics(bp,gw,mark,rp):
    fourPics(bp,gw,mark,rp)
```

fourPics: Combining four pictures

```
def fourPics(bp,gw,mark,rp):
    ''' Returns a new picture with the four
          given pictures in its quadrants
          +---+
          | a b |
          +---+
          | c d |
          +---+
    '''
    def fourPics(a, b, c, d):
        newPic = Layer()
        aHalf = a.clone() # aHalf = a.clone()
        aHalf.scale(0.5) # aHalf.scale(0.5)
        bHalf = b.clone() # bHalf = b.clone()
        bHalf.scale(0.5) # bHalf.scale(0.5)
        cHalf = c.clone() # cHalf = c.clone()
        cHalf.scale(0.5) # cHalf.scale(0.5)
        dHalf = d.clone() # dHalf = d.clone()
        dHalf.scale(0.5) # dHalf.scale(0.5)
        aHalf.move(-50,-50) # aHalf.move(-50,-50)
        bHalf.move(50,50) # bHalf.move(50,50)
        cHalf.move(-50,50) # cHalf.move(-50,50)
        dHalf.move(50,50) # dHalf.move(50,50)
        newPic.add(aHalf)
        newPic.add(bHalf)
        newPic.add(cHalf)
        newPic.add(dHalf)
        return newPic
```

Concepts in this slide:
new operation for pictures: overlay

**fourSame**: Combining four copies of one picture

```python
def fourSame(pic):
    return fourPics(pic, pic, pic, pic)
```

Concepts in this slide: a combinator operation: `fourSame` that creates a new picture.

**Repeated tiling**

```python
def tiling(pic):
    return fourSame(fourSame(fourSame(fourSame(pic))))
```

Concepts in this slide: a combinator operation: `tiling` that creates a new picture.

**How to make a checkerboard?**

```python
checkerboard('black', 'red')
checkerboard('magenta', 'cyan')
```

**DCG on checkerboard**

Hint: Think of the functions `fourSame` and `fourPics`. We’ll solve this problem on the notebook in class.
Combining four rotations of a picture

```python
def rotations(pic):
    return fourPics(clockwise270(pic), pic, clockwise180(pic), clockwise90(pic))

def rotations2(pic):
    return fourPics(pic, clockwise90(pic), clockwise180(pic), clockwise270(pic))
```

A simple recipe for complexity

```python
def wallpaper(pic):
    return rotations(rotations(rotations(rotations(pic))))

def design(pic):
    return rotations2(rotations2(rotations2(rotations2(pic))))
```

A quilt problem

How do we build this complex quilt …

… from simple primitive pictures like this?

Don't focus on programming it with Python. Instead, think about its composition through primitive pictures and combinator functions.

```
triangles('green', 'blue')  patch('red')
```

Divide the quilt into subproblems

A top-down approach for solving problems. Start with the solution and break it into smaller parts.
Conquer the subproblems using “wishful thinking”

Wishful thinking

We suppose that we have a function `quadrant` that can draw each of the four quadrants in this picture.

`.clockwise270(quadrant())`, `.quadrant()`, `.clockwise180(quadrant())`, `.clockwise90(quadrant())`

DCG with Pics 5-25

Glue the subsolutions to solve the problem

Gluing is done through combinator operations such as `fourPics` or `fourSame`.

`.quadrant()`

`.quilt()`

`.def quilt():
    return fourPics(clockwise270(quadrant()), quadrant(), clockwise180(quadrant()), clockwise90(quadrant()))`

DCG with Pics 5-26

Abstracting over the glue

`.def quadrant():
    return corner(star('yellow', 'red', 'blue'),
                  star('red', 'green', 'blue'))`

`.def rotations(pic): # picture function from slide 5-21
    return fourPics(clockwise270(pic), pic, clockwise180(pic), clockwise90(pic))`

`.def quilt():
    return rotations(quadrant())`

To notice:
The function `rotations` takes as argument another function, because calling `quadrant` will return a picture object.

DCG with Pics 5-27

Subproblem: `quadrant()`

Another example of wishful thinking: `star`

`.def quadrant():
    return corner(star('yellow', 'red', 'blue'),
                  star('red', 'green', 'blue'))`

`.def corner(lowerLeftPic, outerPic):
    return fourPics(outerPic, outerPic, lowerLeftPic, outerPic)`
Continue the descent ...

```
def star(innerColor, middleColor, outerColor):
    return rotations(starQuadrant(innerColor, middleColor, outerColor))
```

```
def starQuadrant(squareColor, lowLeftTriColor, upRightTriColor):
    return corner(patch(squareColor),
                   triangles(lowLeftTriColor, upRightTriColor))
```

To notice:
We are using the function `corner` again, which was defined and used in slide 5-28. It captures the pattern of one unique picture in the lower-left corner, surrounded by three copies of another unique picture.

And descend some more ...

```
def star(innerColor, middleColor, outerColor):
    return rotations(starQuadrant(innerColor, middleColor, outerColor))
```

```
def starQuadrant(squareColor, lowLeftTriColor, upRightTriColor):
    return corner(patch(squareColor),
                   triangles(lowLeftTriColor, upRightTriColor))
```

To notice:
We are using the function `corner` again, which was defined and used in slide 5-28. It captures the pattern of one unique picture in the lower-left corner, surrounded by three copies of another unique picture.

... until we reach primitives

```
def quilt():
    return rotations(quadrant())
```

```
def quadrant():
    return corner(star('yellow', 'red', 'blue'),
                  star('red', 'green', 'blue'))
```

```
def star(innerColor, middleColor, outerColor):
    return rotations(starQuadrant(innerColor, middleColor, outerColor))
```

```
def corner(lowerLeftPic, outerPic):
    return fourPics(outerPic, outerPic, lowerLeftPic, outerPic)
```

```
def starQuadrant(squareColor, lowLeftTriColor, urTriColor):
    return corner(patch(squareColor),
                   triangles(lowLeftTriColor, urTriColor))
```

All together now

```
def quilt():
    return rotations(quadrant())
```

```
def quadrant():
    return corner(star('yellow', 'red', 'blue'),
                  star('red', 'green', 'blue'))
```

```
def star(innerColor, middleColor, outerColor):
    return rotations(starQuadrant(innerColor, middleColor, outerColor))
```

```
def corner(lowerLeftPic, outerPic):
    return fourPics(outerPic, outerPic, lowerLeftPic, outerPic)
```

```
def starQuadrant(squareColor, lowLeftTriColor, urTriColor):
    return corner(patch(squareColor),
                   triangles(lowLeftTriColor, urTriColor))
```
Abstracting over quilt colors

How to generalize quilt to define quiltColors?

quiltColors('yellow', 'red', 'green', 'blue')
quiltColors('red', 'blue', 'magenta', 'cyan'))

Challenge

How do we build this picture …

… starting with this primitive?

Test your knowledge

1. Looking at the picture in slide 5-34, describe what would be the “divide” step, the “conquer” step, and the “glue” step.
2. Which of these steps involves “wishful thinking” and for what purpose?
3. Do we need “wishful thinking” once, or many times? On what does it depend?
4. The image primitives we created in the slides 5-6 to 5-9 are all of size 200x200, but when displayed in the canvas window they are much bigger. What is going on?
5. When you compare the solution for the quilt in slide 5-32 to the one you had to write for Bunny Money in PS02 what are some key differences you notice? Why?
6. What does the function fourPics do? Try it out on a piece of paper with some of the primitives on slide 5-7?
7. On a piece of paper create four columns and organize the functions you saw in these slides in categories: functions that create primitive images, transformation operators, combinators, and pattern-capturing. For each of them draw a picture that will remind you what that function does.