CS 1124
Media Computation

Steve Harrison

Lecture 3.2 (September 10, 2008)

Today ...

- Reflecting on Modularity ...
  - **One and only one thing**
  - **Hierarchical decomposition**
- Modifying pixels in a range
  - **mirroring**
  - **mirrorTemple**

Questions on functions

- How can we reuse variable names like **picture** in both a function and in the Command Area?
- Why do we write the functions like this? Would other ways be just as good?
- Is there such a thing as a better or worse function?
- Why don't we just build in calls to **pickAFile** and **makePicture**?

Modularity heuristic: One and only one thing

- We write functions as we do to make them *general* and *reusable*
  - **Programmers hate to have to re-write something they've written before**
  - **They write functions in a general way so that they can be used in many circumstances.**
- What makes a function *general* and thus *reusable*?
  - **A reusable function does One and Only One Thing**

Compare these two programs

```
def makeSunset(picture):
  for p in getPixels(picture):
    value=getBlue(p)
    setBlue(p,value*0.7)
    value=getGreen(p)
    setGreen(p,value*0.7)
```

Yes, they do exactly the same thing!

makeSunset(somepict) has the same effect in both cases

```
def makeSunset(picture):
  reduceBlue(picture)
  reduceGreen(picture)

def reduceBlue(picture):
  for p in getPixels(picture):
    value=getBlue(p)
    setBlue(p,value*0.7)

def reduceGreen(picture):
  for p in getPixels(picture):
    value=getGreen(p)
    setGreen(p,value*0.7)
```

Observations on the new makeSunset

- It's okay to have more than one function in the same Program Area (and file)
- makeSunset in this one is somewhat easier to read.
  - **It's clear what it does "reduceBlue" and "reduceGreen"**
  - **That's important!**

```
def makeSunset(picture):
  reduceBlue(picture)
  reduceGreen(picture)

def reduceBlue(picture):
  for p in getPixels(picture):
    value=getBlue(p)
    setBlue(p,value*0.7)

def reduceGreen(picture):
  for p in getPixels(picture):
    value=getGreen(p)
    setGreen(p,value*0.7)
```

Programs are read by people, not computers!

Considering variations

- We can only do this because **reduceBlue** and **reduceGreen**, do *one and only one thing*.
- If we put **pickAFile** and **makePicture** in them, we'd have to pick a file twice (better be the same file), make the picture—then save the picture so that the next one could get it!

```
def makeSunset(picture):
  reduceBlue(picture)
  reduceGreen(picture)

def reduceBlue(picture):
  for p in getPixels(picture):
    value=getBlue(p)
    setBlue(p,value*0.7)

def reduceGreen(picture):
  for p in getPixels(picture):
    value=getGreen(p)
    setGreen(p,value*0.7)
```

Does makeSunset do one and only one thing?

- Yes, but it's a higher-level, *more abstract* thing.
  - **It's built on lower-level one and only one thing**
- We call this *hierarchical decomposition.*
  - **You have some thing that you want the computer to do?**
  - **Redefine that thing in terms of smaller things**
  - **Repeat until you know how to write the smaller things**
  - **Then write the larger things in terms of the smaller things.**

Are all these pictures the same?

- What if we use this like this in the Command Area:

```
>>> file=pickAFile()
>>> picture=makePicture(file)
>>> makeSunset(picture)
>>> show(picture)
```

```
def makeSunset(picture):
  reduceBlue(picture)
  reduceGreen(picture)

def reduceBlue(picture):
  for p in getPixels(picture):
    value=getBlue(p)
    setBlue(p,value*0.7)

def reduceGreen(picture):
  for p in getPixels(picture):
    value=getGreen(p)
    setGreen(p,value*0.7)
```

What happens when we use a function

- When we type in the Command Area

>>>makeSunset(picture)

Whatever object that is in the *Command Area* variable **picture** becomes the value of the *placeholder (input) variable* **picture** in

```
def makeSunset(picture):
    reduceBlue(picture)
    reduceGreen(picture)
```

makeSunset's picture is then passed as input to **reduceBlue** and **reduceGreen**, but their input variables are completely different from **makeSunset**'s picture.

- **For the life of the functions, they are the same values (picture objects)**

Names have contexts

- In natural language, the same word has different meanings depending on *context*.
  - **Time flies like an arrow**
  - **Fruit flies like a banana**
- A function is its *own* context.
  - **Input variables (placeholders) take on the value of the input values only for the life of the function**
    - Only while it's executing
  - **Variables defined within a function also only exist within the context of that function**
  - **The context of a function is also called its scope**

Input variables are placeholders

- Think of the input variable as a placeholder
  - **It takes the place of the input object**
- During the time that the function is executing, the placeholder variable *stands for* the input object.
- When we modify the placeholder by changing its pixels with **setRed**, we actually change the input object.

Input variables as placeholders (example)

- Imagine we have a wedding computer

```
def marry(husband, wife):
    sayVows(husband)
    sayVows(wife)
    pronounce(husband, wife)
    kiss(husband, wife)
```

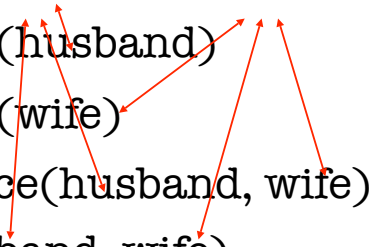
```
def sayVows(speaker):
    print "I, " + speaker + " blah blah"

def pronounce(man, woman):
    print "I now pronounce you..."

def kiss(p1, p2):
    if p1 == p2:
        print "narcissism!"
    if p1 <> p2:
        print p1 + " kisses " + p2
```

So, how do we marry Ben and J.Lo?

Input variables as placeholders (example)

- Imagine we have a wedding computer

```
def marry(husband, wife):
    sayVows(husband)
    sayVows(wife)
    pronounce(husband, wife)
    kiss(husband, wife)
```



```
def sayVows(speaker):
    print "I, " + speaker + " blah blah"

def pronounce(man, woman):
    print "I now pronounce you..."

def kiss(p1, p2):
    if p1 == p2:
        print "narcissism!"
    if p1 <> p2:
        print p1 + " kisses " + p2
```

Input variables as placeholders (example)

- Imagine we have a wedding computer

```
def marry(husband, wife):
    sayVows(husband)
    sayVows(wife)
    pronounce(husband, wife)
    kiss(husband, wife)
```

```
def sayVows(speaker):
    print "I, " + speaker + " blah blah"

def pronounce(man, woman):
    print "I now pronounce you..."

def kiss(p1, p2):
    if p1 == p2:
        print "narcissism!"
    if p1 <> p2:
        print p1 + " kisses " + p2
```

Variables within functions stay within functions

- The variable **value** in **decreaseRed** is created *within* the scope of **decreaseRed**
  - **That means that it only exists while decreaseRed is executing**
- If we tried to *print value* after running decreaseRed, it would work *ONLY* if we already had a variable defined in the Command Area
  - **The name value within decreaseRed doesn't exist outside of that function**
  - **We call that a local variable**

```
def decreaseRed(picture):
  for p in getPixels(picture):
    value=getRed(p)
    setRed(p,value*0.5)
```

Writing real functions

- Functions in the mathematics sense take input and usually return *output*.
  - **Like ord(character) or makePicture(file)**
- What if you create something inside a function that you *do* want to get back to the Command Area?
  - **You can return it.**
  - **We'll talk more about return later—that's how functions output something**

Consider these two functions

```
def decreaseRed(picture):
  for p in getPixels(picture):
    value=getRed(p)
    setRed(p,value*0.5)
```

```
def decreaseRed(picture, amount):
  for p in getPixels(picture):
    value=getRed(p)
    setRed(p,value*amount)
```

- First, it's perfectly okay to have *multiple* inputs to a function.
- The new **decreaseRed** now takes an input of the multiplier for the red value.
  - **decreaseRed(picture,0.5)** would do the same thing
  - **decreaseRed(picture,1.25)** would *increase* red 25%

Names are important

- This function should probably be called **changeRed** because that's what it does.
- Is it more general?
  - **Yes.**
- But is it the one and only one thing that you need done?
  - **If not, then it may be less understandable.**
  - **You can be too general**

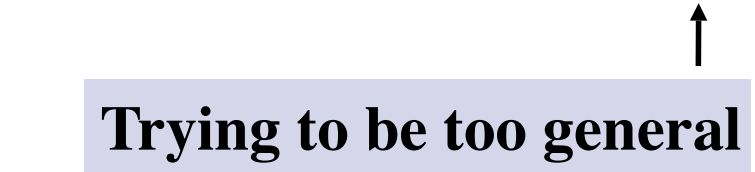
```
def decreaseRed(picture, amount):
  for p in getPixels(picture):
    value=getRed(p)
    setRed(p,value*amount)
```

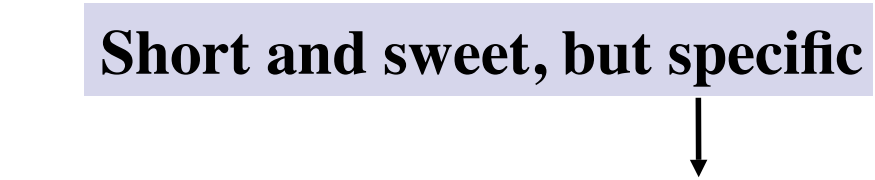
```
def changeRed(picture, amount):
  for p in getPixels(picture):
    value=getRed(p)
    setRed(p,value*amount)
```

Understandability comes first

- Consider these two functions
  - **They do the same thing!**
- The first one *looks like* the other increase/decrease functions we've written.
  - **That may make it more understandable for you to write first.**
- But later, it doesn't make much sense to you
  - **Why multiply by zero? The result is always zero!**
  - **Clearing is a special case of decreasing, so a special function is called for.**

```
def clearBlue(pic):
  for p in getPixels(pic):
    value = getBlue(p)
    setBlue(p,value*0)
```






```
def clearBlue(pic):
  for p in getPixels(pic):
    setBlue(p,0)
```

Understandability comes first

- A couple of other ways to make it understandable
  - **"0" can sometimes be mistaken for "O"**
  - **so writing out "zero" would remind you that you are setting the value to 0**
  - **calling the value "noBlue" would reinforce the idea that you are setting the value of blue to 0 so that there is no blue.**

```
def clearBlue(pic):
  zero = 0
  for p in getPixels(pic):
      setBlue(p,zero)
```

```
def clearBlue(pic):
  noBlue = 0
  for p in getPixels(pic):
      setBlue(p,noBlue)
```

Steps to success heuristic: first make the program easy to understand

- Write your functions so that *you* can understand them *first*
  - **Get your program running**
- ***ONLY THEN*** should you try to make them better
  - **Make them more understandable to other people**
    - E.g. set to zero rather than multiply by zero
    - Another programmer (or you in six months) may not remember or be thinking about increase/decrease functions
  - **Make them more efficient**
    - The new version of **`makeSunset`** (I.e. the one with **`reduceBlue`** and **`reduceGreen`**) takes twice as long as the first version, because it changes all the pixels *twice*
    - But it's easier to understand and to get working in the first place

Today ...

- Reflecting on Modularity ...
  - **One and only one thing**
  - **Hierarchical decomposition**
- Modifying pixels in a range
  - **mirroring**
  - **mirrorTemple**

MirrorVertical

```
def mirrorVertical( source ) :
  mirrorPoint = getWidth(source) / 2
  for y in range(1, getHeight(source)+1 ):
    for xOffset in range(1, mirrorPoint):
      pRight = getPixel(source, xOffset+mirrorPoint, y )
      pLeft = getPixel(source, mirrorPoint-xOffset, y )
      c = getColor(pLeft)
      setColor(pRight, c )
```

- Which side is seen and which side is covered up by the mirroring effect?

Lets change mirrorVertical to mirrorHorizontal

- Transform
  - **vertical -> horizontal**
  - **width -> height**
  - **height -> width**
  - **x -> y**
  - **y -> x**
  - **left -> upper**
  - **right -> lower**
- Since they look so similar is there a way to write a single *general* function that would mirror either horizontally or vertically?

Class projects

- Mostly do in lab section and at home
- Do in groups
- Need some extra credit?
  - **short report on "abstraction"**
    - what is it?
    - What is relation of abstraction in art, poetry, math, computer science?
    - show an example
  - **post to forum**
  - **5 minute presentation in Lab with slides**
  - **worth ONE quiz**

Coming Attractions

- For Friday
  - **Project 2 due**
  - **Extra credit reports on “abstraction” (OPTIONAL)**
- For Monday
  - **(re)Read Chapter 4**
  - **quiz due 10:0 AM**