



COMP2720: Automating Tools for New Media

Making sense of functions



Learning objectives of this lecture

- How to write good functions
- Functions should do only one thing
- Names and their contexts
- Input variables are placeholders
- Scope of variable names
- Readability and understandability are important



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()`?

One and only one thing

- We write functions as we do to make them *general* and *reusable*
 - Programmers hate dislike re-writing something they've written before [the complete Computer Hacker's credo reads like this:
 - **Laziness** (to save energy and labour)
 - **Impatience** (anger when computer is lazy; program such that computer anticipates your need)
 - **Hubris** (excessive pride, feeling of demiurge in you; forces you to write programs which will not besmirch your name)These are programmer's virtues according to **Larry Wall**]
 - 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 into them, we'd have to pick a file twice (better be the same file), make the picture — and 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?
 - Repeat until you know how to write the smaller things.
 - Redefine that *thing* in terms of smaller *things*.
 - Then write the larger things in terms of the smaller things.

Are all these pictures the same?

- What if we use it 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 bananas.
- 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 the function is 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`, for example, we actually change the input object
- Another term for placeholder is *formal parameter*

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..."
```

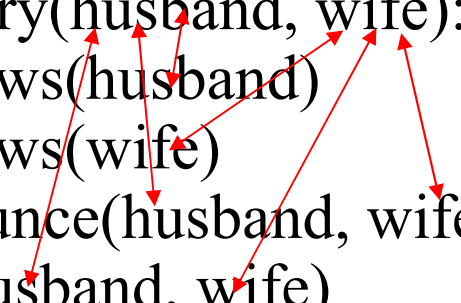
```
def kiss(p1, p2):  
    if (p1 == p2):  
        print "narcissism!"  
    if (p1 <> p2):  
        print p1 + " kisses " + p2
```

So, how do we marry Paul and Mary?

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)
```



```
marry("Paul", "Mary")
```

```
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.gender() == p1.gender()):  
        print "civil union!"  
    if (p1 <> p2):  
        print p1 + " kisses " + p2
```

Input variables as placeholders (example)

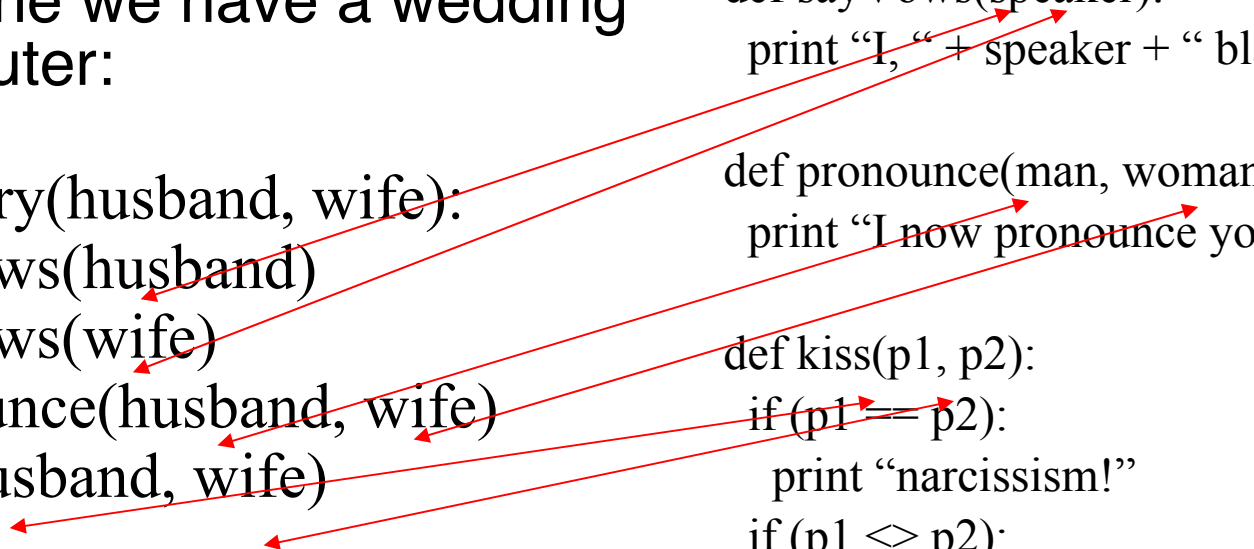
- Imagine we have a wedding computer:

```
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 `value` defined in the Command Area.
 - It would be a different variable `value`!
 - 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 (using the `return` command)
 - That's how functions *output* something.
 - The `return` command must be the last command in a function (any command after `return` will NOT be executed).

Consider these two functions

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

```
def decreaseRed2(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 `decreaseRed2` now takes an input of the multiplier
for the red value.

`decreaseRed2(pic, 0.5)` would do the same thing as `decreaseRed(pic)`
`decreaseRed2(pic, 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 decreaseRed2(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)
```



Trying to be too general

Short and sweet, but specific
low reuse



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

Always make the program easy to understand *first*

- 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.
 - For example, set to zero rather than multiply by zero.
 - Another programmer (or you in three months) may not remember or be thinking about increase/decrease functions.
 - Make them more efficient.
 - The new version of makeSunset (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.

How to program (spatial) slide effects

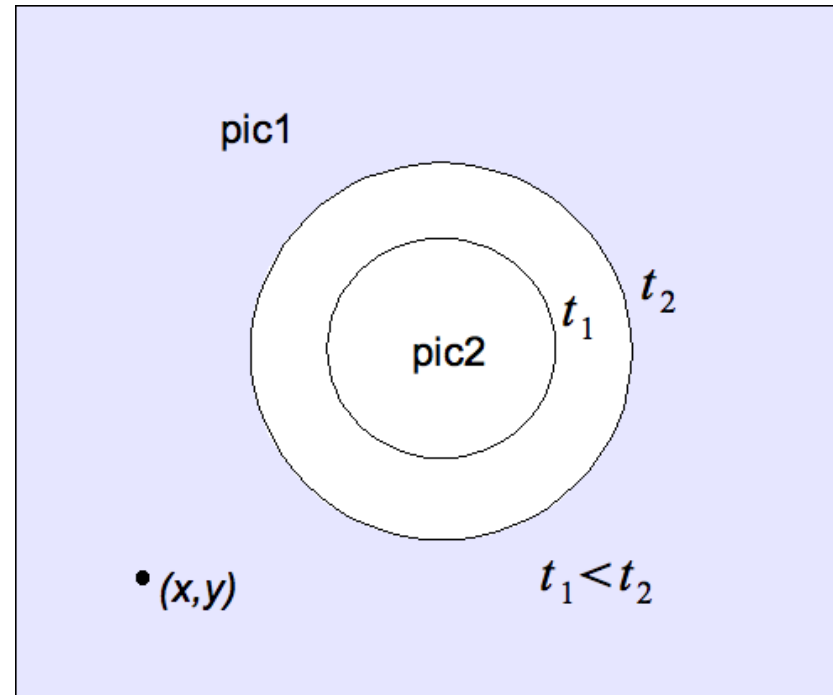
- The slide (transition) effects which we are dealing with in the Assignment 1 are either:
 - spatial --- intermediate transition stages are characterised by presence of both initial picture and final picture, or their parts. The regions occupied by the two components are separated with a well defined boundary (or, boundaries)
 - non-spatial --- when at every transition stage only one image is present albeit in a changing form (in Assignment 1 only Fade In/Out transition is non-spatial)
- How to approach the spatial transitions?
- First, discretise the transition using a controlling parameter (“time”), and consider an arbitrary intermediate stage when there is a clearly defined separation of initial (`pic1`) and final (`pic2`) images. Let's say the chosen stage corresponds to the parameter value t_1

Determine to what picture the pixel belongs?

- The code for setting the values of pixel colours is standard:

```
for x in range(1, w+1):
    for y in range(1, h+1):
        newP =getPixel(canvas,x,y)
        if "(x,y) belongs to pic1" :
            p = getPixel(pic1, $\hat{x}$ , $\hat{y}$ )
        else:
            p = getPixel(pic1, $\tilde{x}$ , $\tilde{y}$ )
        setColor(newP, getColor(p))
```
- The problem is to get this

```
if "(x,y) belongs to pic1"
```
- It can be a tricky problem if the regions occupied by pic1 and pic2 are complex 2D areas, **but** for Assignment 1 transitions the problem is manageable
- Why (\hat{x}, \hat{y}) and (\tilde{x}, \tilde{y}) not just (x, y) ?





What to do now

- Work on assignment 1!
- Read section 3.4.1 in text book (on functions)