



# COMP2720: Automating Tools for New Media

“Do we have to write all those  
lines?”

Functional programming



# Want to write fewer lines of code?

- You can write fewer lines of code and get the same programs written, if you're willing to *trust* your functions.
- When you really understand functions, you can do all kinds of amazing things in very few lines of code.
  - Use functions that apply functions to data.
  - Use recursion: Have functions that call themselves (next lecture).

# Functions are just values associated with names

- We *call* a function by stating its name followed by inputs in parentheses.
- Without parentheses, the name of the function still has a value.
  - It's the function!
- Functions are also data.
  - They can be used as input to other functions!

```
>>> print makeSamplePage
<function makeSamplePage at 4222078>
>>> print fileEntry
<function fileEntry at 10206598>
```

# Introducing apply

- `apply` takes a function as input and the inputs to that function in a sequence.
- `apply` literally applies the function to the input.

```
def hello(someone):  
    print "Hello,",someone  
>>> hello("Mark")  
Hello, Mark  
>>> apply(hello,["Mark"])  
Hello, Mark  
>>> apply(hello,["Betty"])  
Hello, Betty
```

## More useful: map

- map is a function that takes as input a function and a sequence.
- But it applies the function to each input in the sequence, and returns whatever the function returns for each.

```
>>> map(hello,
        ["Mark", "Betty", "Matthew",
         "Jenny"])
Hello, Mark
Hello, Betty
Hello, Matthew
Hello, Jenny
[None, None, None, None]
```



**filter:** Returns those for whom the function is true.

- filter also takes a function and a sequence as input.
- It applies the function to each element of the sequence.
  - If the return value of the function is true (1), then filter returns that element.
  - If the return value of the function is false (0), then filter skips that element.

## filter example

```
def rname(somename):  
    if somename.find("r") == -1:  
        return 0 # False  
    if somename.find("r") != -1:  
        return 1 # True
```

```
>>> rname("January")  
1  
>>> rname("July")  
0  
>>> filter(rname,  
           ["Mark", "Betty", "Matthew",  
            "Jenny"])  
['Mark']
```



# We can make `rname` shorter using a logical operator

- An expression like `somename.find("r") == -1` actually does evaluate to 0 (*False*) or 1 (*True*).
- There are operations we can perform on logical values.
  - Just like `+` is an operation on numbers and strings.
- One of these is `not`.
  - It creates the opposite of whatever the input value is, *true* or *false*.

# Making rname shorter

```
def rname2(somename):
```

```
    return not(somename.find("r") == -1)
```

```
>>> filter(rname2, ["Mark", "Betty", "Matthew", "Jenny"])
```

```
['Mark']
```

## reduce: Combine the results

- reduce takes a function and a sequence, like the others.
- But reduce combines the results.
- In this example, we total all the numbers by adding  $1+2$ , then  $(1+2) + 3$ , then  $(1+2+3)+4$ , then  $(1+2+3+4)+5$

```
def add(a,b):  
    return a+b  
>>> reduce(add,[1,2,3,4,5])  
15
```

# Do we really need to define add?

- Turns out that we don't even have to give a function a name to be able to use it.
- A name-less function is called a lambda.
  - It's an old name, that actually dates back to one of the very oldest programming languages, *Lisp*.
- Wherever you'd use a function name, you can just stick in a lambda.
  - lambda takes the input variables, colon (:), and the body of the function (usually just a single line, or else you'd want to name the function.)

# Using lambda

```
>>> reduce(lambda a,b: a+b, [1,2,3,4,5])
```

```
15
```

- This actually does the *exact* same thing as:

```
def add(a,b):
```

```
    return a+b
```

```
>>> reduce(add,[1,2,3,4,5])
```

```
15
```

# Defining *factorial* with reduce and lambda

- Remember factorial (n!) from math class:
  - Factorial of  $n$  is:  $n! = n * n-1 * n-2 * \dots * 1$

```
def factorial(a):
```

```
    return reduce(lambda a,b:a*b, range(1,a+1))
```

```
>>> factorial(2)
```

```
2
```

```
>>> factorial(3)
```

```
6
```

```
>>> factorial(4)
```

```
24
```

```
>>> factorial(10)
```

```
3628800
```

# Why did we learn about apply?

- map and filter (as we'll soon see) can really be used to implement real programs.
- Why apply?
  - Because that's how map and filter are implemented!
  - Given apply, you can do your own.

```
def mymap(function,list):  
    for i in list:  
        apply(function, [i])
```

```
>>> mymap(hello,  
          ["Fred","Barney","Wilma","  
          Betty"])  
Hello, Fred  
Hello, Barney  
Hello, Wilma  
Hello, Betty
```



# Interesting... but useful? Yes!

- These are really interesting ideas:
  - Functions are *data* that can be used as *inputs*.
  - We can create *functions that manipulate functions*.
    - Meta-functions?
- This is *functional programming*.
  - The style (*paradigm*) we've been doing so-far is called *procedural*.
    - We define processes: *Procedures*.



# Functional programming

- Functional programming is about using layers of functions and functions that apply functions to solve problems.
- It's a powerful form of programming, allowing you to do a lot in very few lines of code.
- Functional programming is particularly useful in *artificial intelligence* (AI) research and in building prototypes of systems.
  - These are both places where the problems are hard and ill-defined, so you want to get as far as possible with as few lines as possible.

# Making turnRed functional

- Remember this?

```
def turnRed():
    brown = makeColor(57,16,8)
    file = r"C:\Documents and Settings\Mark Guzdial\My
        Documents\mediasources\barbara.jpg"
    picture=makePicture(file)
    for px in getPixels(picture):
        color = getColor(px)
        if distance(color,brown)<100.0:
            redness=getRed(px)*1.5
            setRed(px,redness)
    show(picture)
    return(picture)
```

# Let's make it a functional paradigm program

```
def checkPixel(apixel):  
    brown = makeColor(57,16,8)  
    return  
        distance(getColor(apixel),brown)<100.0
```

```
def turnRed(apixel):  
    setRed(apixel,getRed(apixel)*1.5)
```

- For comparison:

```
def turnRed():  
    brown = makeColor(57,16,8)  
    file = r"C:\Documents and Settings\Mark  
        Guzdial\My  
        Documents\mediasources\barbara.jpg"  
    picture=makePicture(file)  
    for px in getPixels(picture):  
        color = getColor(px)  
        if distance(color,brown)<100.0:  
            redness=getRed(px)*1.5  
            setRed(px,redness)  
    show(picture)  
    return(picture)
```

## It's now just a one line program

- What we want to do is filter out pixels that match `checkPixel`, then map the function `turnRed` to that result.

```
map(turnRed, filter(checkPixel, getPixels(pic)))
```

# Really using the one-liner

```
>>> pic=makePicture( getMediaPath("barbara.jpg"))  
>>> map(turnRed, filter(checkPixel, getPixels(pic)))
```

- Exercise for the interested student:
  - Rewrite this function with *just* lambda's! Then it *really is* just a single line of code!



# A new way of thinking

- In functional programming, you don't write functions with big loops that process all the data.
- Instead, you write small functions that process one piece of the data.
  - Then you apply the small function to all the data, using things like `map` and `filter`.
- You end up writing fewer lines of code for solving problems.