Functions and Scope

What happens when the following code is run?

In [1]:
```
var1 = 10;

function testfun1()
    var1 = 20;
    return 0
end
```

Out[1]:
```
testfun1 (generic function with 1 method)
```

In [2]:
```
var1
```

Out[2]:
```
10
```

In [3]:
```
testfun1()
```

Out[3]:
```
0
```

In [4]:
```
var1
```

Out[4]:
```
10
```

Julia functions in general do not mutate the state of variables. So here even though inside the function there is also a variable with the same name, it does not change the outside value.

If you want to change the outside value:

In [5]:
```
function testfun2!()
    global var1
    var1 = 20;
    return 0
end
```

Out[5]:
```
testfun2! (generic function with 1 method)
```

In [6]:
```
testfun2!()
```

Out[6]:
```
0
```

In [7]:
```
var1
```

Out[7]:
```
20
```

The convention with julia is that mutating functions end in an `!`. This is true for other standard julia library functions too (e.g. `push!()`, etc.)

Conditionals

In [8]:
```
function isOdd( x::Int64 )

    if x % 2 == 1
        return true
    else
        return false
    end

end
```

Out[8]:
```
isOdd (generic function with 1 method)
```

In [9]:
```
isOdd(4)
```

Out[9]:
```
false
```

Let's try with more cases. How would you code up the function:

$$f(x) = \begin{cases} 0 & x = 0 \\ \cos(x) & x > 0 \\ -1 - x & x < 0 \end{cases}$$

In [10]:
```
function func2( x )

    if x == 0
        return 0
    elseif x > 0
        return cos(x)
    else
        return -1-x
    end

end
```

Out[10]:
```
func2 (generic function with 1 method)
```

However let's try something:

In [11]:
```
function func3( x )

    #Quadratic polynomial with roots at 0.1, -4.3
    return x^2 + 4.2x - 0.43

end
```

Out[11]:
```
func3 (generic function with 1 method)
```

In [12]:
```
func3(0.1)
```

Out[12]:
```
5.551115123125783e-17
```

In [13]:
```
func2(func3(0.1))
```

Out[13]:
```
1.0
```

What happened? This is a problem with cutoff, floating numbers in computers in general are not stored exactly - they are cutoff after a certain number of digits. This can introduce errors in computing as we see here. How can we fix this?

In [14]:
```
function func2fix( x )

    if abs(x) < 1e-12
        return 0
    elseif x > 1e-12
        return cos(x)
    else
        return -1-x
    end

end
```

Out[14]:
```
func2fix (generic function with 1 method)
```

In [15]:
```
func2fix(func3(0.1))
```

Out[15]:
```
0
```

This is a common trick in mathematical computing, sometimes to get the right answer you have to introduce a small error into the function due to numerical inaccuracies.

Boolean operations

An important tool when it comes to conditionals are the boolean operations `&&`, `||`, `~` (AND, OR, NOT)

In [16]:
```
true && true
```

Out[16]:
```
true
```

In [17]:
```
true && false
```

Out[17]:
```
false
```

In [18]:
```
true || false
```

Out[18]:
```
true
```

In [19]:
```
function isPositiveEven(x)

    if x % 2 == 0 && x > 0
       return true
    end
    return false

end
```

Out[19]:
```
isPositiveEven (generic function with 1 method)
```

In [20]:
```
isPositiveEven(-4)
```

Out[20]:
```
false
```

In [21]:
```
isPositiveEven(4)
```

Out[21]:
```
true
```

For loops and While loops

This is how iteration works in most languages including Julia. If you come from taking CS61A, I know they teach you recursion as a prime tool for doing this. I have **very** strong opinions on this (which is also actually somewhat shared in industry) in that recursion is almost always the worst way to do iteration. This is because it is memory intensive and often far far slower than a standard for or while loop.

In [22]:
```
function func4( x )

    tot = 0;
    for i = 1:x
        tot += x;
    end
    return tot

end
```

Out[22]:
```
func4 (generic function with 1 method)
```

In [23]:
```
func4( 10 )
```

Out[23]:
```
100
```

For loops and while loops are largely equivalent, use whichever is more convenient for the task at hand:

In [24]:
```
function func5bad( x )
    tot = 0; counter = 0;
    while counter < x
       tot += x;
    end
    return tot
end
```

Out[24]:
```
func5bad (generic function with 1 method)
```

Why is the above function bad? The `counter` variable is never updated, so the while loop never stops - this is called an infinite loop. How to fix this?

In [25]:
```
function func5( x )
    tot = 0; counter = 0;
    while counter < x
        tot += x;
        counter += 1;
    end
    return tot
end
```

Out[25]:
```
func5 (generic function with 1 method)
```

In [26]:
```
func5(10)
```

Out[26]:
```
100
```

Practice Question!

Predict the outcome of the following code:

In [27]:
```
function weirdFn( x,y )

    for i = 1:5
        x /= 2;
        y -= 3;
        if x < 2 || y < 0
           return x,y
        end
    end
    return x,y

end
```

Out[27]:
```
weirdFn (generic function with 1 method)
```

In [28]:
```
weirdFn( 16,100 )
```

Out[28]:
```
(1.0, 88)
```

In [29]:
```
weirdFn( 16,8 )
```

Out[29]:
```
(2.0, -1)
```

Write a function to generate the nth Fibonacci number: (1,1,2,3,5,8,...) the next number is sum of previous two numbers

In [30]:
```
function fibonaccihelper( a,b )

    return b, a+b

end

function fibonacci( x )

    if x == 1 || x == 2
        return 1
    end

    a = 1; b = 1;
    for i = 2:x
        a,b = fibonaccihelper( a,b );
    end

    return b

end
```

Out[30]:
```
fibonacci (generic function with 1 method)
```

In [31]:
```
fibonacci(5)
```

Out[31]:
```
8
```

In [ ]:
```

```