In [92]:
```
# WEEK 2: Control flow and functions
```

In [93]:
```
#Control flow and functions
```

In [94]:
```
#Compound Expressions
z = begin
           x = 1
           y = 2
           x + y
    end
```

Out[94]: 3

In [95]:
```
#Since these are fairly small, simple expressions, they could easily be p
laced onto a single line, which is where the ; chain syntax comes in hand
y
z = (x = 1; y = 2; x + y)
```

Out[95]: 3

In [96]:
```
#This syntax is particularly useful with the terse single-line function d
efinition form introduced in Functions. Although it is typical, there is
no requirement that begin blocks be multiline or that ; chains be single-
line:
(x = 1;
y = 2;
x + y)
```

Out[96]: 3

In [97]:
```
#Conditional Evaluation
#Conditional evaluation allows portions of code to be evaluated or not ev
aluated depending on the value of a boolean expression. Here is the anato
my of the if-elseif-else conditional syntax:
if x < y
    println("x is less than y")
elseif x > y
    println("x is greater than y")
else
    println("x is equal to y")
end
```

```
x is less than y
```

In [98]:
```
#If the condition expression x < y is true, then the corresponding block
is evaluated; otherwise the condition expression x > y is evaluated, and
if it is true, the corresponding block is evaluated; if neither expressio
n is true, the else block is evaluated. Here it is in action:
function test(x, y)
    if x < y
         println("x is less than y")
            elseif x > y
               println("x is greater than y")
           else
               println("x is equal to y")
           end

end
```

Out[98]: test (generic function with 1 method)

In [99]:
```
test(1, 2)
```

x is less than y

In [100]:
```
test(2, 1)
```

x is greater than y

In [101]:
```
test(1, 1)
```

x is equal to y

In [102]:
```
#The elseif and else blocks are optional, and as many elseif blocks as de
sired can be used. The condition expressions in the if-elseif-else constr
uct are evaluated until the first one evaluates to true, after which the
associated block is evaluated, and no further condition expressions or bl
ocks are evaluated.

#if blocks are "leaky", i.e. they do not introduce a local scope. This me
ans that new variables defined inside the if clauses can be used after th
e if block, even if they weren't defined before. So, we could have define
d the test function above as
     function test(x,y)
           if x < y
               relation = "less than"
           elseif x == y
               relation = "equal to"
           else
               relation = "greater than"
           end
           println("x is ", relation, " y.")
       end
```

Out[102]: test (generic function with 1 method)

In [103]:
```
test(2, 1)
```

x is greater than y.

In [104]:
```
#The variable relation is declared inside the if block, but used outside. However, when depending on this behavior, make sure all possible code paths define a value for the variable. The following change to the above function results in a runtime error
    function test(x,y)
           if x < y
               relation = "less than"
           elseif x == y
               relation = "equal to"
           end
           println("x is ", relation, " y.")
       end
```

Out[104]:
```
test (generic function with 1 method)
```

In [105]:
```
test(1,2)
```

```
x is less than y.
```

In [106]:
```
test(2,1)
```

```
UndefVarError: `relation` not defined

Stacktrace:
 [1] test(x::Int64, y::Int64)
   @ Main .\In[104]:8
 [2] top-level scope
   @ In[106]:1
```

In [107]:
```
#if blocks also return a value, which may seem unintuitive to users coming from many other languages. This value is simply the return value of the last executed statement in the branch that was chosen, so
x = 3

if x > 0
 "positive!"
else
"negative..."
end
```

Out[107]:
```
"positive!"
```

In [108]:
```
#Short-Circuit Evaluation
#The && and || operators in Julia correspond to logical “and” and “or” op
erations, respectively, and are typically used for this purpose. However,
they have an additional property of short-circuit evaluation: they don't
necessarily evaluate their second argument, as explained below. (There ar
e also bitwise & and | operators that can be used as logical “and” and “o
r” without short-circuit behavior, but beware that & and | have higher pr
ecedence than && and || for evaluation order.)
#Short-circuit evaluation is quite similar to conditional evaluation. The
behavior is found in most imperative programming languages having the &&
and || boolean operators: in a series of boolean expressions connected by
these operators, only the minimum number of expressions are evaluated as
are necessary to determine the final boolean value of the entire chain. S
ome languages (like Python) refer to them as and (&&) and or (||). Explic
itly, this means that:
    #In the expression a && b, the subexpression b is only evaluated if a
evaluates to true
    #In the expression a || b, the subexpression b is only evaluated if a
evaluates to false.
#The reasoning is that a && b must be false if a is false, regardless of
the value of b, and likewise, the value of a || b must be true if a is tr
ue, regardless of the value of b. Both && and || associate to the right,
but && has higher precedence than || does. It's easy to experiment with t
his behavior:
 t(x) = (println(x); true)
```

Out[108]: t (generic function with 1 method)

In [109]:
```
f(x) = (println(x); false)
```

Out[109]: f (generic function with 2 methods)

In [110]:
```
t(1) && t(2)
```

```
1
2
```

Out[110]: true

In [111]:
```
t(1) && f(2)
```

```
1
2
```

Out[111]: false

In [112]:
```
f(1) && t(2)
```

```
1
```

Out[112]: false

In [113]:
```
f(1) && f(2)
```

```
1
```

Out[113]: false

In [114]:
```
t(1) || t(2)
```

```
1
```

Out[114]:
```
true
```

In [115]:
```
t(1) || f(2)
```

```
1
```

Out[115]:
```
true
```

In [116]:
```
f(1) || t(2)
```

```
1
2
```

Out[116]:
```
true
```

In [117]:
```
f(1) || f(2)
```

```
1
2
```

Out[117]:
```
false
```

In [118]:
```
#You can easily experiment in the same way with the associativity and pre
cedence of various combinations of && and || operators.
#This behavior is frequently used in Julia to form an alternative to very
short if statements. Instead of if <cond> <statement> end, one can write
<cond> && <statement> (which could be read as: <cond> and then <statement
>). Similarly, instead of if ! <cond> <statement> end, one can write <con
d> || <statement> (which could be read as: <cond> or else <statement>).
    #For example, a recursive factorial routine could be defined like thi
s:
        function fact(n::Int)
           n >= 0 || error("n must be non-negative")
           n == 0 && return 1
           n * fact(n-1)
       end
```

Out[118]:
```
fact (generic function with 1 method)
```

In [119]:
```
fact(5)
```

Out[119]:
```
120
```

In [120]:
```
fact(0)
```

Out[120]:
```
1
```

In [121]:
```
#Boolean operations without short-circuit evaluation can be done with the bitwise boolean operators introduced in Mathematical Operations and Elementary Functions: & and |. These are normal functions, which happen to support infix operator syntax, but always evaluate their arguments:
f(1) & t(2)
```

```
1
2
```

Out[121]: false

In [122]:
```
t(1) | t(2)
```

```
1
2
```

Out[122]: true

In [123]:
```
#On the other hand, any type of expression can be used at the end of a conditional chain. It will be evaluated and returned depending on the preceding conditionals:
true && (x = (1, 2, 3))
```

Out[123]: (1, 2, 3)

In [124]:
```
false && (x = (1, 2, 3))
```

Out[124]: false

In [125]:
```
#Repeated Evaluation: Loops
#There are two constructs for repeated evaluation of expressions: the while loop and the for loop. Here is an example of a while loop
i = 1;

     while i <= 3
           println(i)
           global i += 1
       end
```

```
1
2
3
```

In [126]:
```
#Here the 1:3 is a range object, representing the sequence of numbers 1, 2, 3. The for loop iterates through these values, assigning each one in turn to the variable i. One rather important distinction between the previous while loop form and the for loop form is the scope during which the variable is visible. A for loop always introduces a new iteration variable in its body, regardless of whether a variable of the same name exists in the enclosing scope. This implies that on the one hand i need not be declared before the loop. On the other hand it will not be visible outside the loop, nor will an outside variable of the same name be affected. You'll either need a new interactive session instance or a different variable name to test this:
  for j = 1:3
           println(j)
       end
```

```
1
2
3
```

In [127]:
```
j
```

Out[127]:
```
0
```

In [128]:
```
j=0
for j = 1:3
           println(j)
       end
```

```
1
2
3
```

In [129]:
```
#Use for outer to modify the latter behavior and reuse an existing local variable.

#See Scope of Variables for a detailed explanation of variable scope, outer, and how it works in Julia.

#In general, the for loop construct can iterate over any container. In these cases, the alternative (but fully equivalent) keyword in or ∈ is typically used instead of =, since it makes the code read more clearly:
for i in [1,4,0]
           println(i)
       end
```

```
1
4
0
```

In [130]:
```
for s ∈ ["foo","bar","baz"]
           println(s)
       end
```

```
foo
bar
baz
```

In [131]:
```
#Various types of iterable containers will be introduced and discussed in
later sections of the manual (see, e.g., Multi-dimensional Arrays).

#It is sometimes convenient to terminate the repetition of a while before
the test condition is falsified or stop iterating in a for loop before th
e end of the iterable object is reached. This can be accomplished with th
e break keyword:
i = 1;

 while true
           println(i)
           if i >= 3
               break
           end
           global i += 1
       end
```

```
1
2
3
```

In [132]:
```
for j = 1:1000
           println(j)
           if j >= 3
               break
           end
       end
```

```
1
2
3
```

In [133]:
```
#Without the break keyword, the above while loop would never terminate on
its own, and the for loop would iterate up to 1000. These loops are both
exited early by using break.

#In other circumstances, it is handy to be able to stop an iteration and
move on to the next one immediately. The continue keyword accomplishes th
is:
for i = 1:10
           if i % 3 != 0
               continue
           end
           println(i)
       end
```

```
3
6
9
```

In [134]:
```
#This is a somewhat contrived example since we could produce the same beh
avior more clearly by negating the condition and placing the println call
inside the if block. In realistic usage there is more code to be evaluate
d after the continue, and often there are multiple points from which one
calls continue.

#Multiple nested for loops can be combined into a single outer loop, form
ing the cartesian product of its iterables:
for i = 1:2, j = 3:4
           println((i, j))
       end
```

```
(1, 3)
(1, 4)
(2, 3)
(2, 4)
```

In [135]:
```
#With this syntax, iterables may still refer to outer loop variables; e.
g. for i = 1:n, j = 1:i is valid. However a break statement inside such a
loop exits the entire nest of loops, not just the inner one. Both variabl
es (i and j) are set to their current iteration values each time the inne
r loop runs. Therefore, assignments to i will not be visible to subsequen
t iterations:
for i = 1:2, j = 3:4
           println((i, j))
           i = 0
       end
```

```
(1, 3)
(1, 4)
(2, 3)
(2, 4)
```

In [136]:
```
# If this example were rewritten to use a for keyword for each variable,
then the output would be different: the second and fourth values would co
ntain 0.

#Multiple containers can be iterated over at the same time in a single fo
r loop using zip:
for (j, k) in zip([1 2 3], [4 5 6 7])
           println((j,k))
       end

#Using zip will create an iterator that is a tuple containing the subiter
ators for the containers passed to it. The zip iterator will iterate over
all subiterators in order, choosing the

#ith element of each subiterator in the  ith iteration of the for loop. O
nce any of the subiterators run out, the for loop will stop
```

```
(1, 4)
(2, 5)
(3, 6)
```

In [137]:
```
#=Functions
In Julia, a function is an object that maps a tuple of argument values to
a return value.
Julia functions are not pure mathematical functions, because they can alt
er and be affected by the global state of the program.
The basic syntax for defining functions in Julia is: =#
```

In [138]:
```
function f(x,y)
           x + y
 end
```

Out[138]: f (generic function with 2 methods)

In [139]:
```
f(x,y) = x + y
```

Out[139]: f (generic function with 2 methods)

In [140]:
```
#The above function accepts two arguments x and y and returns the value o
f the last expression evaluated, which is x + y.

#There is a second, more terse syntax for defining a function in Julia.
#The traditional function declaration syntax demonstrated above is equiva
lent to the following compact "assignment form":
f(x,y) = x + y
```

Out[140]: f (generic function with 2 methods)

In [141]:
```
f(2,3)
```

Out[141]: 5

In [142]:
```
#Without parentheses, the expression f refers to the function object, and
can be passed around like any other value:
g = f;
g(2,3)
```

Out[142]: 5

In [143]:
```
#As with variables, Unicode can also be used for function names:
∑(x,y) = x + y
```

Out[143]: ∑ (generic function with 1 method)

In [144]:
```
∑(2, 3)
```

Out[144]: 5

In [145]:
```
#Argument Passing Behavior
#= Julia function arguments follow a convention sometimes called "pass-by
-sharing", which means that values are not copied when they are passed to
functions. Function arguments themselves act as new variable bindings (ne
w "names" that can refer to values), much like assignments argument_name
= argument_value, so that the objects they refer to are identical to the
passed values. Modifications to mutable values (such as Arrays) made with
in a function will be visible to the caller. (This is the same behavior f
ound in Scheme, most Lisps, Python, Ruby and Perl, among other dynamic la
nguages.)

For example, in the function=#
function f(x, y)
    x[1] = 42    # mutates x
    y = 7 + y    # new binding for y, no mutation
    return y
end
```

Out[145]:
```
f (generic function with 2 methods)
```

In [146]:
```
#= The statement x[1] = 42 mutates the object x, and hence this change wi
ll be visible in the array passed by the caller for this argument.
On the other hand, the assignment y = 7 + y changes the binding ("name")
y to refer to a new value 7 + y, rather than mutating the original object
referred to by y, and hence does not change the corresponding argument pa
ssed by the caller.
This can be seen if we call f(x, y):
=#
a = [4,5,6]
```

Out[146]:
```
3-element Vector{Int64}:
 4
 5
 6
```

In [147]:
```
b = 3
f(a, b) # returns 7 + b == 10
```

Out[147]:
```
10
```

In [148]:
```
a  # a[1] is changed to 42 by f
```

Out[148]:
```
3-element Vector{Int64}:
 42
  5
  6
```

In [149]:
```
b  # not changed
#=
As a common convention in Julia (not a syntactic requirement), such a fun
ction would typically be named f!(x, y) rather
than
f(x, y), as a visual reminder at the call site that at least one of the a
rguments (often the first one) is being mutated.=#
```

Out[149]:
```
3
```

In [150]:
```
#=Argument-type declarations
You can declare the types of function arguments by appending ::TypeName t
o the argument name, as usual for Type Declarations in Julia.
For example, the following function computes Fibonacci numbers recursivel
y: =#
fib(n::Integer) = n ≤ 2 ? one(n) : fib(n-1) + fib(n-2)
#and the ::Integer specification means that it will only be callable when
n is a subtype of the abstract Integer type.
```

Out[150]: `fib (generic function with 1 method)`

In [151]:
```
#= The return Keyword
The value returned by a function is the value of the last expression eval
uated, which, by default,
is the last expression in the body of the function definition.
In the example function, f, from the previous section this is the value o
f the expression x + y. As an alternative, as in many other languages,
the return keyword causes a function to return immediately, providing an
expression whose value is returned: =#
function Q(x,y)
    return x * y
    x + y
end
```

Out[151]: `Q (generic function with 1 method)`

In [152]:
```
f(x,y) = x + y
```

Out[152]: `f (generic function with 2 methods)`

In [153]:
```
function M(x,y)
    return x * y
    x + y
end
```

Out[153]: `M (generic function with 1 method)`

In [154]:
```
f(2,3)
```

Out[154]: `5`

In [155]:
```
M(2,3)
```

Out[155]: `6`

In [156]:
```
#Of course, in a purely linear function body like g, the usage of return
is pointless since the expression x + y is never evaluated and we could s
imply make x * y the last expression in the function and omit the return.
In conjunction with other control flow, however, return is of real use.
#Here, for example, is a function that computes the hypotenuse length of
a right triangle with sides of length x and y, avoiding overflow:
function hypot(x,y)
           x = abs(x)
           y = abs(y)
           if x > y
               r = y/x
               return x*sqrt(1+r*r)
           end
           if y == 0
               return zero(x)
           end
           r = x/y
           return y*sqrt(1+r*r)
       end
```

Out[156]: hypot (generic function with 1 method)

In [157]:
```
hypot(3, 4)
```

Out[157]: 5.0

In [158]:
```
#= Return type
A return type can be specified in the function declaration using the :: o
perator. This converts the return value to the specified type.
=#
function S(x, y)::Int8
           return x * y
       end;
```

In [159]:
```
typeof(S(1, 2))
#This function will always return an Int8 regardless of the types of x an
d y. See Type Declarations for more on return types.
```

Out[159]: Int8

In [160]:
```
#= Returning nothing
For functions that do not need to return a value (functions used only for
some side effects), the Julia convention is to return the value nothing:
=#
function printx(x)
    println("x = $x")
    return nothing
end
```

Out[160]: printx (generic function with 1 method)

In [161]:
```
#This is a convention in the sense that nothing is not a Julia keyword bu
t only a singleton object of type Nothing. Also, you may notice that the
printx function example above is contrived, because println already retur
ns nothing, so that the return line is redundant.
```

In [ ]:
```

```