# Lecture 4: Control Flow and Loops

Math 98

# Agenda

- Relations (review)
- Logical statements
- Boolean expressions
- `if-else` statements
  - Exercises
- `for` loops
  - Exercises
- `while` loops
  - `break`
  - Exercises

# Relations (review)

The following statements will take value 0 (if false) or 1 (if true)

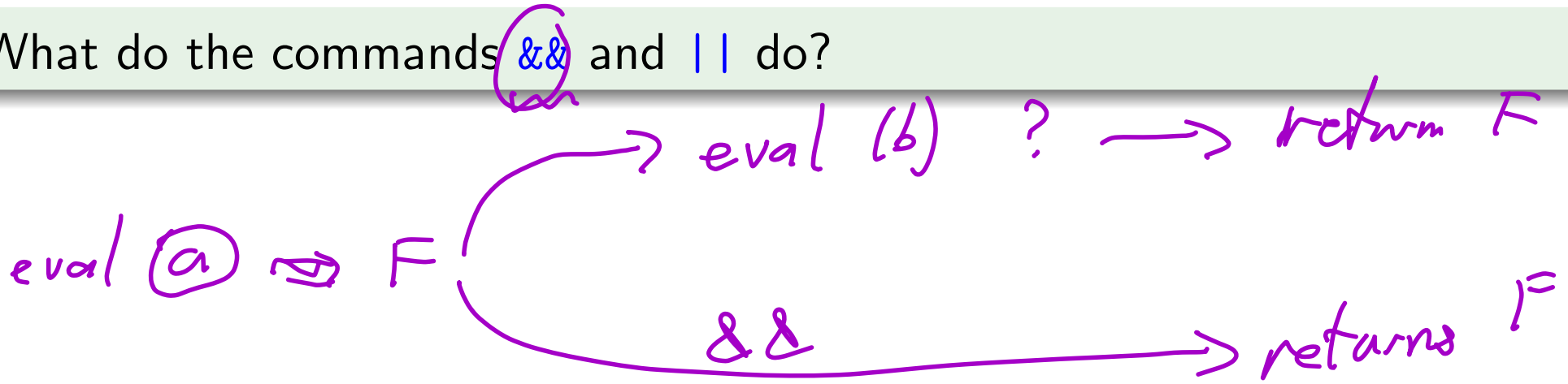
- $a < b$: $a$ less than $b$
- $a > b$: $a$ greater than $b$
- $a <= b$: $a$ less than or equal to $b$
- $a >= b$: $a$ greater than or equal to $b$
- $a == b$: $a$ equal to $b$ (note the doubled equals sign!)
- $a \sim= b$: $a$ not equal to $b$

# Logical Statements

a && b

- `and(a,b)` or equivalently `a & b`
- `or(a,b)` or equivalently `a | b`
- `not(a)` equivalently ~a
- `xor(a,b)`

What do the commands `&&` and `||` do?



# Boolean Expressions

A boolean expression is any expression involving relations or logical statements:

$$((4 <= 100)|(-2 > 5))\&(\texttt{true}|\sim\texttt{false})$$

Boolean expressions evaluate to 1 for true and 0 for false. Note that 0 and 1 are just numbers and are not in a separate class for logicals.

```
>> 5 + true
ans =
     6
```

The order of operations is as follows:

1. negation
2. relations
3. and
4. or

# if-else Statements: General Structure

This construct is used where the decision to execute one or another set of computations depends on the value of a boolean expression.

```
if   this boolean expression is true
          execute these commands
elseif   this second expression is true instead
          then execute these other commands
else
          do this if those earlier conditions are false
end
```

# if-else Statements: Example 1

What does this return?

```
if 4 > 3
        disp('first one!')
elseif pi == 3.14
        disp('second one!')
else
        disp('neither were true!')
end
```

# if-else Statements: Example 2

What does this return?

```
if 4 < 3
            disp('first one!')
elseif pi == 3.14
            disp('second one!')
else
            disp('neither were true!')
end
```

# if-else Statements: Example 3

What does this return?

```
if 4 < 3
        disp('first one!')
elseif pi == 3.14
        disp('second one!')
elseif false
        disp('third one!')
end
```

# if-else Statements: Example 3(b)

What's wrong with this?

```
if 4 < 3
        disp('first one!')
elseif pi == 3.14
        disp('second one!')
elseif
        disp('third one!')
end
```

# Exercise: `comparison.m`

1

Write a script that prompts the user for two numbers (call them $x$ and $y$). It should output `The numbers are equal` if $x = y$ and `The numbers are not equal` otherwise.

2 $x > y$ or $< y$ or $= y$

3 ↓ next exercise

# Exercise: `quadroots.m`

Write a script that prompts the user for three integers `a`, `b`, `c`. These are the coefficients to the quadratic $p(x) = ax^2 + bx + c$. Display a message saying whether the quadratic has 1) distinct real roots, 2) a repeated root, or 3) complex roots.

# `for` Loops: Motivation

Is $n$ prime?

- Try dividing $n$ by 2,3,...
- If no smaller number divides $n$, then $n$ is prime

We need a way to run multiple tests, one after the other.
We also need the function `mod()`, which finds remainders after division:

```
>> mod(17,5)
ans =
     2
>> mod(33,3)
ans =
     0
```

# for Loops: Description

Used to repeat a set of commands a certain number of times

[1, 2, 3, ..., NOT]

```
for countVariable = 1 : numberOfIterations

    % do something here
    % this part will run
    % (numberOfIterations) times

end
```

# for Loops: Example

Simple Example:

```
>> for i = 1:4
     i + 2
end
ans =
     3
ans =
     4
ans =
     5
ans =
     6
```

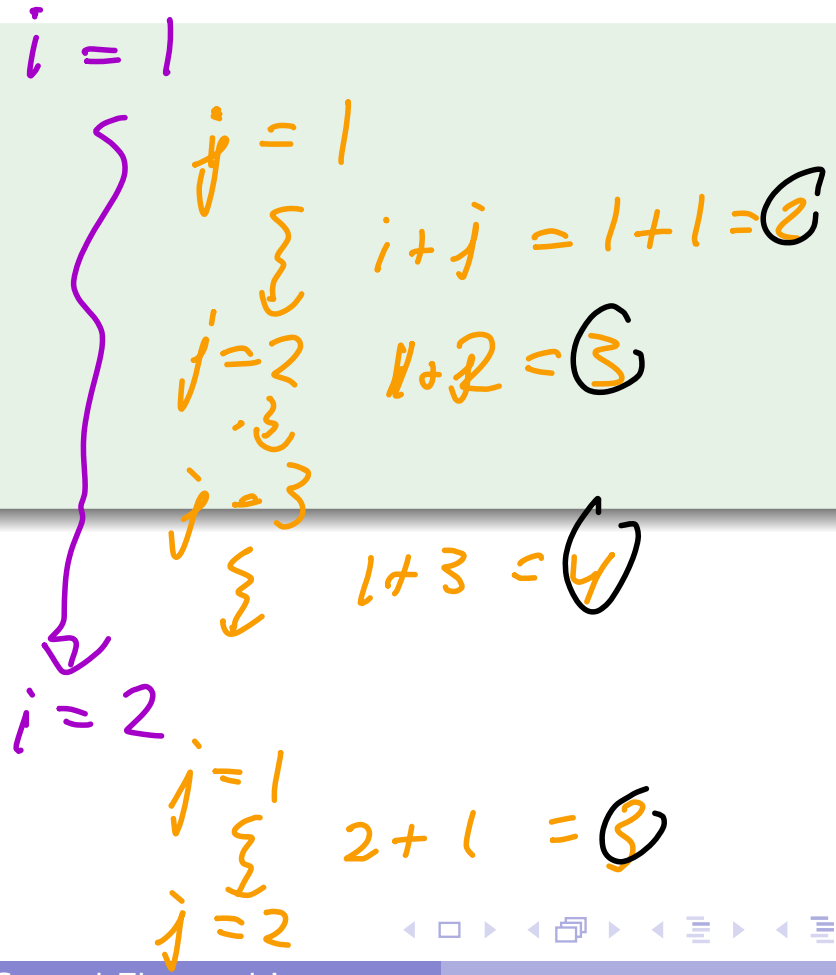
[1, 2, 3, 4]

i = 1
  i+2 = 1+2 = 3
i = 2
  i+2 = 2+2 = 4
i = 3
  3+2 = 5
i = 4
  4+2 = 6
end

# Nested `for` Loops: Example

Here is a for loop within a for loop. This is called a nested loop.

```
for i = 1:4
    for j = 1:3
        i+j
    end
end
```



# Exercise: sumCubes.m

sum(v.^3)

Write a program `sumCubes.m` of the form

```
function S = sumCubes(v)
```

that takes a vector as input and returns the sum of the cubes of its elements. For pedagogical purposes, do this by:

1. Initializing a variable `S = 0` to keep track of the sum
2. Use a `for` loop

Do you know a much simpler way to do this?

# Example: `testPrime.m`

Write a function of the form

```
function [isPrime,divisor] = testPrime(n)
```

that takes in an integer $n$ and returns isPrime = true if $n$ is prime and false otherwise. It should return divisor = NaN if the integer is prime and its smallest divisor otherwise.

(This should be obvious, but don't use the built in `MATLAB` function `isprime`)

# while Loops: Introduction

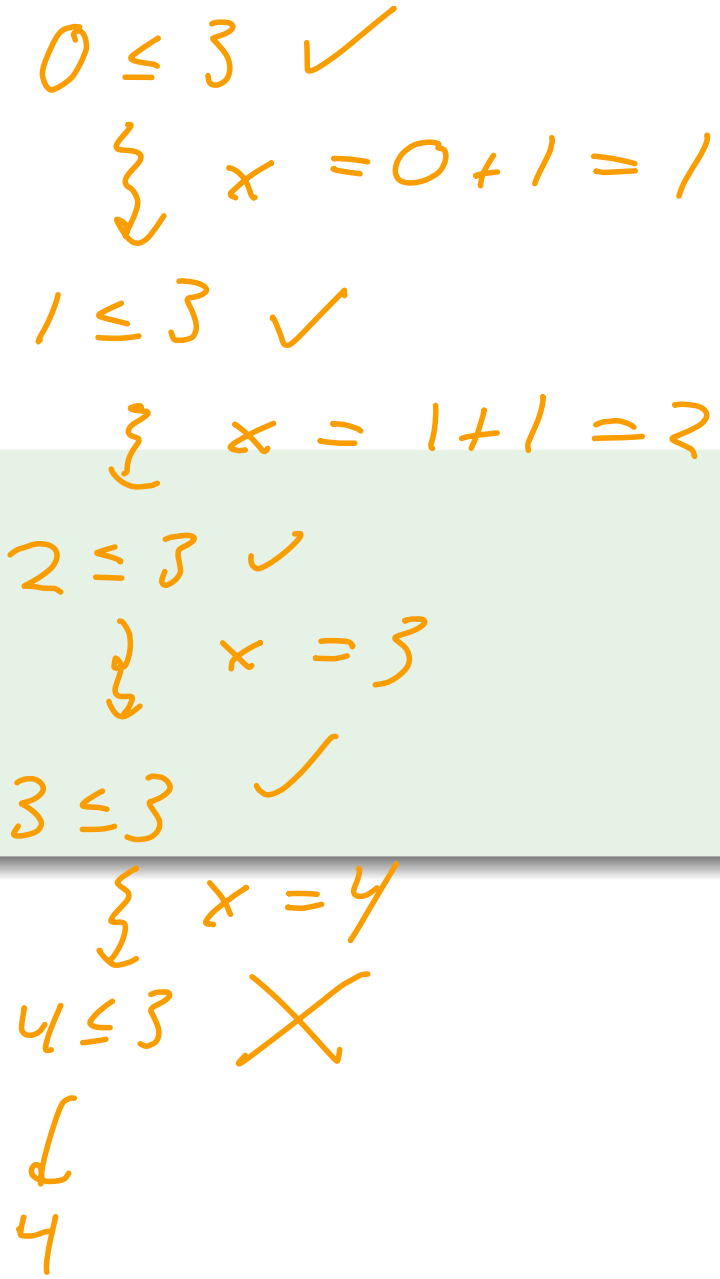
A statement to repeat a section of code *until* some condition is satisfied.

```
while [EXPRESSION is true]

    % repeat this part until
    % (EXPRESSION) is false
    % be sure to modify (EXPRESSION) in this loop

end
```

# while Loops: Example

Here is a simple example.

```
x = 0;
while x<=3
    x = x+1;
end
```



# while Loops: Nontermination

A for loop does "stuff" for a set number of times. A while loop does "stuff" until some condition is no longer satisfied. This may go on forever!

```
x = 0;
while x<=3
    x = x-1;
end
```

# while Loops: continue

In both `for` and `while` loops, `continue` skips to the next run of the loop.

```
for i = 0:3:30
    if mod(i,2) == 0
        continue
    end
    fprintf('%d ', i);
end
```

It's often possible to avoid using `continue` by restructuring your code. Can you do that with the code above?

# while Loops: break

The command break terminates the loop.

```
while true
    guess = input('What number am I thinking of? ');
    if guess == 5
        fprintf('Lucky guess \n');
        break
    else
        fprintf('WRONG');
    end
end
```

Can you rewrite this code so that it doesn't use break?

# `while` Loops: In Class Demo

Demonstration of `while`, `continue`, and `break`: `manyFrogs.m`

# Exercise: bisection.m

Implement a MATLAB function bisection.m of the form

```
function p = bisection(f, a, b, tol)
% f: function handle y = f(x)
% a: Beginning of interval [a, b]
% b: End of interval [a, b]
% tol: user provided tolerance for interval width

% p: approximation to the root
```

# Exercise: newton.m

Implement a function newton.m of the form

```
function p = newton(f, df, p0, tol)
% f: function handle y = f(x)
% df: function handle of derivative y' = f'(x)
% p0: initial estimate of the root
% tol: user provided tolerance for accuracy of solution

% p: approximation to the root
```