Logic, loops and control flow

NENS 230
Eddy Albarran
10.6.15
Announcements

• Eddy’s office hours this week:
  – Weds 11AM-12:30PM
  – Email to schedule other time(s)
Outline for today

• Review of relational operators
• Logic and branching
• Loops
• Advanced control flow

The cornerstone to programming!
Relational operators

Test if relationship is true or false

- Less than
  \[2 < 3\] (true)
- Greater than
  \[2 > 3\] (false)
- Less than or equal to
  \[1 \leq 1\] (true)
- Greater than or equal to
  \[4 \geq 5\] (false)
- Exactly equal
  \[8 = 2^3\] (true)
- Not equal
  \[\pi \approx 3.14\] (true)

Output is datatype called logical
Remember...

```
“=” means assignment
>> x = 5
ans =
  5

“==” tests equality
>> x = 5;
>> x == 5
ans =
  1
```

One of the most common programming mistakes ever

You will make it. Many times.
Relational operators are *vectorized*

Operate on all members of an array simultaneously

```plaintext
>> x = 1:10

ans =
    1  2  3  4  5  6  7  8  9  10

>> x > 5

ans =
    0  0  0  0  0  0  0  1  1  1  1  1  1  1
```
Relational operators are *vectorized*

Operate on all members of an array simultaneously

```matlab
>> x = 1:10
ans =
     1     2     3     4     5     6     7     8     9    10

>> x > 5
ans =
    0     0     0     0     0     1     1     1     1     1
```
Logical indexing

• Allows selection of any elements meeting criteria
• Criteria usually defined by relational operators

```matlab
>> x = 1:10;
>> idx = x > 5;
>> x(idx)
```

```plaintext
ans =
   6    7    8    9   10
```

Usually combine in one expression: `x(x > 5)`
Assignment by logical indexing

```matlab
>> x = -2:2;
>> x(x < 0) = 0  \% Truncate values at 0

ans =
    0  0  0  1  2

>> x(x < 0) = [] \% Remove negatives

ans =
    0  1  2
```
Logical indexing and `find`

`find` returns indices of nonzero elements

```plaintext
>> x = 1:10;
>> x > 5
ans =
   0  0  0  0  0  1  1  1  1  1

>> find(x > 5)
ans =
    6  7  8  9  10
```
A comment on the `find` function

```matlab
>> data = [0, 0, 1, 0, 1, ..., 1, 0];
>> ltime = avg_logical_time(data);
>> ftime = avg_find_time(data);
>> ftime / ltime

ans =
    7.5426
```

Logical indexing is faster by an order of magnitude
When to use \texttt{find}

\begin{itemize}
\item Need the actual indices of a condition
\item Need to know only first place condition is satisfied
\item Better for arrays with many zeros (sparse)
\end{itemize}

\textit{Everything else should use logical indexing}
Other useful logical functions

- isempty
- isnan, isinf
- isfinite  \% true if not +/-Inf, NaN
- isnumeric, ischar, iscell, isstruct
Our code thus far

do_thing1(data);
do_thing2(data);
do_thing3(data);
.
.
.
do_thing73(data);

All code executed unconditionally

But what if we only want to do_thing2() if data meets some criterion?
Outline for today

- Relational operators
- Logic and branching
- Loops
- Advanced control flow
Branching

• Execute blocks of code only if certain conditions are met
• Each possible path is a “branch”

```java
if condition1 % keyword to start block
    do_something();
end % keyword to end block
```
Branching: if/else

```plaintext
if p_value <= significance_threshold
  % jump here if condition is true
  keep_data();
else
  % jump here if condition is false
  % keep_data is NOT executed!
  ignore_data();
end
```
Branching: multiple conditions

```python
if condition1
    do_thing1();
end
if condition2
    do_thing2();
end
if condition3
    do_thing3();
end
```
Branching: `if/elseif`

Test multiple conditions in a single block

```plaintext
if condition1
    do_thing1();
elseif condition2
    do_thing2();
elseif condition3
    do_thing3();
...
else
    do_last Thing();
end
```

One and *only* one of these functions will be executed.
Branching with many conditions

• Used `if/elseif` blocks to test many conditions
• Becomes cumbersome and hard to read with a large number of conditions
Branching: \texttt{switch/case}

\begin{verbatim}
switch \texttt{my\_var} \\
  \textbf{case 1} \\
    \texttt{do\_thing1();} \\
  \textbf{case 2} \\
    \texttt{do\_thing2();} \\
  \textbf{otherwise} \\
    \texttt{do\_default\_thing();} \\
end
\end{verbatim}

Equivalent to:

\begin{verbatim}
if \texttt{my\_var == 1}
\end{verbatim}

Default code block
if (mod(i, 3) == 0) && (mod(i, 5) == 0)
    fprintf('%d div. by 3 and 5\n', i);
elseif mod(i, 3) == 0
    fprintf('%d div. by 3\n', i);
elseif mod(i, 5) == 0
    fprintf('%d div. by 5\n', i);
else
    fprintf('%d div. by neither\n', i);
end
Branching: if/else

```matlab
for i = 1:100
    if mod(i, 2) == 0
        fprintf(' %d is even\n', i);
    else
        fprintf(' %d is odd\n', i);
    end
end
```
Branching: `switch` with strings

```python
s = input('Are you happy?', 's');
switch s
  case {'y', 'yes'}
    awesome();
  case {'y', 'yes'}
    too_bad();
  otherwise
    stock_answer();
end
```
Combining logical values

Use “&” or “|” to combine arrays of logicals

```plaintext
>> val1 = [0, 1, 1, 0];
>> val2 = [1, 1, 0, 0];
>> result = val1 & val2

result =
  0  1  0  0  0
```

Compares two arrays of same size, element-wise
Combining logical values

Use "&&" or "||" to combine scalar logicals

```plaintext
>> 0 && 1
ans =
  0

>> 0 || 1
ans =
  1
```
## What is truth?

<table>
<thead>
<tr>
<th>Value</th>
<th>((\textit{if } w) \text{ == } ?)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(w = -1;)</td>
<td>\text{true}</td>
</tr>
<tr>
<td>(x = 0.5;)</td>
<td>\text{true}</td>
</tr>
<tr>
<td>(y = \text{`a string';})</td>
<td>\text{true}</td>
</tr>
<tr>
<td>(z = \text{Inf;})</td>
<td>\text{true}</td>
</tr>
<tr>
<td>(a = 0;)</td>
<td>\text{false}</td>
</tr>
<tr>
<td>(b = [0, 1];)</td>
<td>?</td>
</tr>
</tbody>
</table>

Any scalar value except 0 evaluates to true

Arrays evaluate to true if \textit{all} elements are true

Generally do direct tests:

\((\textit{if } w \text{ == } -1)\)
Our code thus far

```python
>> data1 = load_data(file1);
>> process_data(data1);
>> data2 = load_data(file2);
>> process_data(data2)
.
.
.
>> process_data(data87)
>> publish_results();
```

Loops to the rescue
Outline for today

• Relational operators
• Logic and branching
• Loops
• Advanced control flow
Loops

Allow repeated execution of a block of code

For loop:
```
for i = N:M
    do_something;
end
```

While loop:
```
while condition
    do_something;
end
```
Loops

>> for i = 1:5
fprintf(""i" is now \%d\n", i);
end
"i" is now 1
"i" is now 2
"i" is now 3
"i" is now 4
"i" is now 5
>>
Loops

>> i = 1;
>> while i <= 5
fprintf(""i" is now %d\n", i);
i = i + 1;
end
"i" is now 1
"i" is now 2
"i" is now 3
"i" is now 4
"i" is now 5
>>
Loops: index variables

for i = 1:n
    for j = 1:m
        for k = 1:b
            get_grade(i, j, k);
        end
    end
end

Difficult to understand code’s purpose from loop variable names
Loops: index variables

```plaintext
for si = 1:num_students
    for ai = 1:num_assignments
        for pi = 1:num_problems
            get_grade(si, ai, pi);
        end
    end
end
```

Better
Loops: index variables

```plaintext
for student = 1:num_students
    for assgn = 1:num_assignments
        for problem = 1:num_problems
            get_grade(student, assgn, ... problem);
        end
    end
end
```

Best!
Which loop type?

We can use a `for` loop wherever can use a `while` loop, and vice versa.

**for**
- Use to repeat code a predetermined number of times
- Automatic tracking of index variable

**while**
- Use to repeat code as long as condition is true
- Automatic tracking of condition’s truth value
Which loop type?

| start = 10;  |
| fact = 1;    |
| **for** ni = start:-1:2 |
|   fact = fact * ni; |
| **end**     |

| start = 10; |
| ni = start; |
| fact = 1;   |
| **while** ni > 1 |
|   fact = fact * number; |
|   ni = ni - 1; |
| **end**     |
Outline for today

- Relational operators
- Logic and branching
- Loops
- Advanced control flow
Control flow

Methods for fine-grained control of loops

1. Combining loops and branching
2. `continue`: Skip to next loop iteration
3. `break`: Exit loop altogether
Control flow: loops and branching

```matlab
for i = 1:N
    % check condition on each
    % loop iteration
    if condition
        do_this();
    else
        do_that();
    end
end
```
Control flow: loops and branching

```matlab
for i = 1:100
    if is_even(i)
        fprintf('%%d is even\n', i);
    else
        fprintf('%%d is odd\n', i);
    end
end
```
Control flow: **continue**
Skip to next loop iteration

```matlab
for i = 1:num_datasets
    if (~meets_criteria(i))
        continue;  % skips the rest of % the loop, **but still** % increments i
    else
        process_data(i);
    end
end
```
Control flow: break

Exit loop altogether

```plaintext
for i = 1:num_datasets
    if (meets_criteria(i))
        break; % abort the loop as soon
        % as we find valid data
    end
end
% break moves us here
process_data(i);
```
Control flow: `break`

Continuous loops

```matlab
while 1 % loop forever
    new_data = get_more_data();
    if isempty(new_data)
        break; % no more data, exit loop
    end
    process_data(new_data);
end
```
Problem set 3

Förster resonance energy transfer (FRET)

1. Two nearby light-sensitive molecules, chromophores
2. Excited chromophore may donate energy to neighbor
3. Energy transfer proportional to distance
Problem set 3


\[
\frac{F_A}{F_A + F_D}
\]
Review

• Branching
  – *if*: execute code if condition true
  – *else*: execute code if condition false
  – *elseif & switch/case*: test multiple statements

• Loops
  – *for*: execute block defined number of times
  – *while*: execute block as long as condition true

• Control flow
  – combine loops and branching
  – *continue*: skip this loop iteration, start next
  – *break*: exit loop altogether