# PSYCH 20A (PROF. FRANE) – LECTURE 7
# ifs, loops, and switches

## Using `if`

Sometimes we want to do something only if some condition is met. That's where `if` statements come in handy. For example, the following code prompts the user to answer a question, and then outputs "Correct!" only if the correct answer is given:

```
capitalResponse = input('What is the capital of California? ', 's') ;

if strcmpi(capitalResponse, 'Sacramento') == 1
   fprintf('\nCorrect!\n\n')
end
```

Whatever code comes between the `if` statement and the `end` is performed only if `capitalResponse` matches the character string 'Sacramento'. The convention is to indent that code for clarity.

In the above example, we would probably also want to output "Incorrect!" if the answer is incorrect. We could do that by adding another if statement, like this:

```
capitalResponse = input('What is the capital of California? ', 's') ;

if strcmpi(capitalResponse, 'Sacramento') == 1
   fprintf('\nCorrect!\n\n')
end

if strcmpi(capitalResponse, 'Sacramento') == 0
   fprintf('\nIncorrect!\n\n')
end
```

But because those two if statements involve mutually exclusive events, a more efficient method would be to use an `else` statement. We can think of `else` as meaning "otherwise" or "in any other case." Whatever code comes between the `if` statement and the `else` is performed under one condition, and whatever code comes between the `else` and the `end` happens in any other case. For example:

```
capitalResponse = input('What is the capital of California? ', 's') ;

if strcmpi(capitalResponse, 'Sacramento') == 1
   fprintf('\nCorrect!\n\n')
else
   fprintf('\nIncorrect!\n\n')
end
```

We can also include an `elseif` statement. The following code outputs "Correct!" if the answer is correct, outputs "No response" if the user just hits return without answering, and outputs "Incorrect!" in any other case:

```
capitalResponse = input('What is the capital of California? ', 's') ;

if strcmpi(capitalResponse, 'Sacramento') == 1
   fprintf('\nCorrect!\n\n')
elseif strcmp(capitalResponse, '') == 1
   fprintf('\nNo response\n\n')
else
   fprintf('\nIncorrect!\n\n')
end
```

# For-loops

When we want to do the same procedure repeatedly a specified number of times, we can use a *for-loop*. For example, let's say we want to do the following procedure 3 times: Prompt the user to input a number, and then report the square of that number. We can do that as follows:

```
for i = 1:3
   inputNum = input('Enter a number: ') ;
   fprintf(['\nYour number squared is ' num2str(inputNum^2) '\n\n'])
end
```

All the lines of code between the `for` statement and the `end` are what get repeated. The convention is to indent that code for clarity.

Notice the row-vector `1:3` in the `for` statement. The number of elements in that vector (in this case, 3) determines the number of times we go through the loop. Thus, we could have used `4:6` or `[0 0 0]` or `'abc'` or any other 3-element row-vector and gotten the same behavior.

You may wonder what the `i` is about. The `i` is a variable whose value is updated each time we go through the loop. The values that `i` takes on are the values in the vector in the `for` statement. Thus, the first time we go through the loop, `i` is set to the first value in the vector, which is 1. The second time we go through the loop, `i` is set to the second value in the vector, which is 2. And the last time we go through the loop, `i` is set to the last value in the vector, which is 3.

In the above example, `i` isn't actually used for anything. But what if we wanted to store all of the user-entered numbers in a vector? In that case, we could use the `i` to index the vector where we're storing those numbers, as in the following example:

```
inputNum = NaN(1, 3) ; % initialize vector to be filled by entered numbers

for i = 1:3
   inputNum(i) = input('Enter a number: ') ;
   fprintf('\nYour number squared is %d.\n\n', inputNum^2)
end
```

The following example uses the for-loop variable both for indexing the vector to be filled and for indexing a cell array containing character-strings used in the prompt:

```
category = {'color', 'food', 'city'} ; % array of categories
response = cell(1, 3) ; % initialize cell array to be filled by text responses

for i = 1:numel(wordCategory)
   response{i} = input(['What is your favorite ' category{i} '? '], 's') ;
end
```

**Naming the for-loop variable**
Instead of `i`, typically you should use a more descriptive name for the for-loop variable. A recommended practice is to use the letter `i` followed by whatever it is you're counting. For example, in the above example, naming the variable something like `iCategory` would be preferable.

**Using a matrix in the for-statement**
In rare instances, we might want to enter a matrix, instead of a row-vector, in the `for` statement, e.g.: `for i = [1 2 3 ; 10 20 30]`. In that case, the `i` would be set to a column-vector each time through the loop: [1 ; 10] the first time, [2 ; 20] the second time, and [3 ; 30] the last time.

# While-loops

Instead of doing a procedure a prespecified number of times, we can keep repeating something indefinitely until some condition is met. For example, we could repeatedly ask the user a question until the correct answer is given. The following code repeatedly prompts the user to guess a color until the correct color is guessed:

```
colorGuess = '' ; % initialize color guess as empty character-vector

while strcmpi(colorGuess, 'purple') == 0
   colorGuess = input('\nTry to guess my favorite color: ', 's') ;
end

fprintf('\nCorrect!\n\n')
```

A `while` statement is a type of logical statement that tells us under what condition we stay in the loop. In this case, it says to keep repeating the loop as long as the variable `colorGuess` does not match the character-vector `'purple'` (remember that "==0" is like saying "is false").

Notice that the `fprintf` command that outputs "Correct!" is *after* the end of the loop, so it doesn't happen until the loop is escaped (i.e., until the correct color is guessed).

Note that we had to initialize `colorGuess` before the `while` statement. Otherwise, the `while` statement would produce an error because it would refer to a variable that didn't exist yet.

**Counting the number of times through the loop**
Here's the same code as above, but we add a "counter" called `numGuesses` that keeps a tally of how many tries it takes the user to guess correctly (i.e., how many times we go through the loop before escaping):

```
colorGuess = '' ; % initialize color guess as empty character-vector
numGuesses = 0  ; % initialize counter that tallies the number of guesses

while strcmpi(colorGuess, 'purple') == 0 % stay in loop until purple guessed
   numGuesses = numGuesses + 1 ; % update number of guesses
   colorGuess = input('\nTry to guess my favorite color: ', 's') ;
end

fprintf('\nCorrect!\n\n')
```

If we want to record the guesses themselves, rather than just the *number* of guesses, we can use the counter to index an array of responses (which we call `guessVector` here):

```
currentGuess = '' ; % initialize current guess as empty character vector
numGuesses   = 0  ; % initialize counter that tallies the number of guesses
guessVector  = cell(1, 1) ; % initialize cell array to fill with guesses

while strcmpi(currentGuess, 'purple') == 0 % stay in loop until guess purple
   currentGuess = input('\nTry to guess my favorite color: ', 's') ;
   numGuesses   = numGuesses + 1        ; % update number of guesses
   guessVector{numGuesses} = colorGuess ; % add current guess to vector
end

fprintf('\nCorrect!\n\n')
```

## If, elseif, and while-statements with multiple conditions

Like any logical statement, an `if`, `elseif`, or `while` statement can include `&` or | operators. But you may notice that if you include an `&` or | operator in an `if`, `elseif`, or `while` statement in a script, Matlab will highlight the operator and suggest that you replace it with an `&&` or || operator, respectively. Although the single `&` and | operators will typically have the same result, the `&&` and || operators are slightly more efficient for Matlab to execute.

To modify one of our earlier color-guessing examples to accept either "purple" or "violet" as correct:

```
colorGuess = '' ; % initialize color guess as empty character-vector

while strcmpi(colorGuess, 'purple') == 0  &&  strcmpi(colorGuess, 'violet') == 0
   colorGuess = input('\nTry to guess my favorite color: ', 's') ;
end

fprintf('\nCorrect!\n\n')
```

In fact, any time you're asking AND or OR questions that produce a *single* true/false answer, you can use the `&&` and || operators. For example, if `x` is a single value, 5, then typing `x > 1 && x < 10` will output `1` (meaning "true"). But if `x` is a vector of *multiple* values, then you have to use the single `&` and | operators, and Matlab will output a vector of trues (1s) and falses (0s).

Note that to escape the while-loop in one condition OR another, the while-statement must contain an AND, because the while statement defines the conditions in which you stay in the while loop. Conversely, to escape the while-loop only when one condition AND another are both true, the while loop must contain an OR. Remember, "if AND gets you out, then OR keeps you in," and "if OR gets you out, then AND keeps you in." To use a real-life example, imagine you only want to leave your house if you have your phone AND your keys. In that case, you want to stay in your house WHILE the opposite is true: either you don't have your phone OR you don't have your keys. Now imagine you only want to leave your house if it's not raining OR you have an umbrella. In that case, you want to stay in your house WHILE the opposite is true: it's raining AND you have no umbrella.

## Indentation

Part of writing neat, well-organized programs is using proper indentation. The `if` statement and its corresponding `end` statement (and any corresponding `else` and `elseif` statements) should be exactly lined up with each other, and all the lines in-between should be indented relative to those statements. Similarly, in a for-loop or while-loop, the `for` or `while` statement should be exactly lined up with then corresponding `end`, and all the lines in-between should be indented relative to those statements.

Often there are loops or `if` constructions "nested" inside each other, in which case there are multiple levels of indentation. Proper indentation is especially important in such cases to make it clear "what's inside of what."

To have Matlab automatically fix your indentation in a program, highlight the code you want to fix and click on the green "smart indent" icon next to the word "Indent" at the top of the screen. Alternatively, you can use the smart-indent hotkey (`CTRL-i` on a PC, or `COMMAND-i` on a Mac).

## Logical shorthand

When we say `if x` without any further operators (such as == or <=), it's implied that we mean `if x ~= 0` (or in other words, "if x is true"). The same applies to `elseif x` and `while x`.

Thus, in the first example in this handout, we could replace this line:

```
if strcmpi(capitalResponse, 'Sacramento') == 1
```

with this more concise line:

```
if strcmpi(capitalResponse, 'Sacramento')
```

When we say `if ~x` without any further operators (such as == or <=), it's implied that we mean `if x == 0` (or in other words, "if x is false"). The same applies to `elseif ~x` and `while ~x`.

Thus, in our while-loop examples, we could replace this line:

```
while strcmpi(colorGuess, 'purple') == 0
```

with this more concise line:

```
while ~strcmpi(colorGuess, 'purple')
```

## Interrupting the flow of a loop

There are three functions that can be used to interrupt the normal flow of a for-loop or while-loop:

**`continue`** immediately skips to the next iteration of the loop, without executing any more lines of code for the current iteration.

**`break`** immediately exits the current loop entirely, without executing any more lines of code in the loop at all. In a for-loop, `break` will abort the loop even if the vector in the for-statement hasn't been exhausted (e.g., the for-statement says to go from 1 to 10 and you're only on 4). In a while-loop, `break` will abort the loop even if the condition in the while-statement is true (e.g., the while-statement says `while x==2` and x is indeed 2). When the `break` command is in an inner loop embedded inside an outer loop, only the inner loop is aborted.

**`return`** immediately exits not only out of a loop, but also out of the entire program. `return` can be used anywhere in a program, not just inside a loop. If the `return` command appears in a function being called by another program, the `return` command will only exit out of the function, not out of the program that uses the function.

## Using `switch`

In certain situations, we can use a `switch` construction instead of an `if` construction. That can be handy when using an `if` statement would require several ORs (e.g., if variable x equals 'A' or 'B' or 'C' or 'D' we do one thing, and if variable x equals 'E' or 'F' or 'G' we do something else). First we define the "switch" (the variable we're evaluating), and then we define how to respond in each "case" (each specific set of possible values the variable could have). The variable we're evaluating must be either a single value (e.g., a number or string) or a character vector. Here's an example:

```
day = input('Enter a day of the week:', 's') ;

switch day
   case {'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday'}
      fprintf('\nThat is a school day.\n')
   case {'Saturday', 'Sunday'}
      fprintf('\nThat is not a school day.\n')
   otherwise
      fprintf('\nDay not recognized.\n')
end
```

The first `case` statement functions like an `if` statement, and any subsequent `case` statements function like `elseif` statements. In each `case` statement, we're asking if the switch variable matches any of the elements in the case vector. And of course, `otherwise` functions like an `else`. Thus, the above code is a slightly more efficient version of the following:

```
day = input('Enter a day of the week:', 's') ;

if     strcmp(day, 'Monday'   ) || strcmp(day, 'Tuesday' ) || ...
       strcmp(day, 'Wednesday') || strcmp(day, 'Thursday') || ...
       strcmp(day, 'Friday'   )
       fprintf('\nThat is a school day.\n')

elseif strcmp(day, 'Saturday') || strcmp(day, 'Sunday')
       fprintf('\nThat is not a school day.\n')
else
       fprintf('\nDay not recognized.\n')
end
```