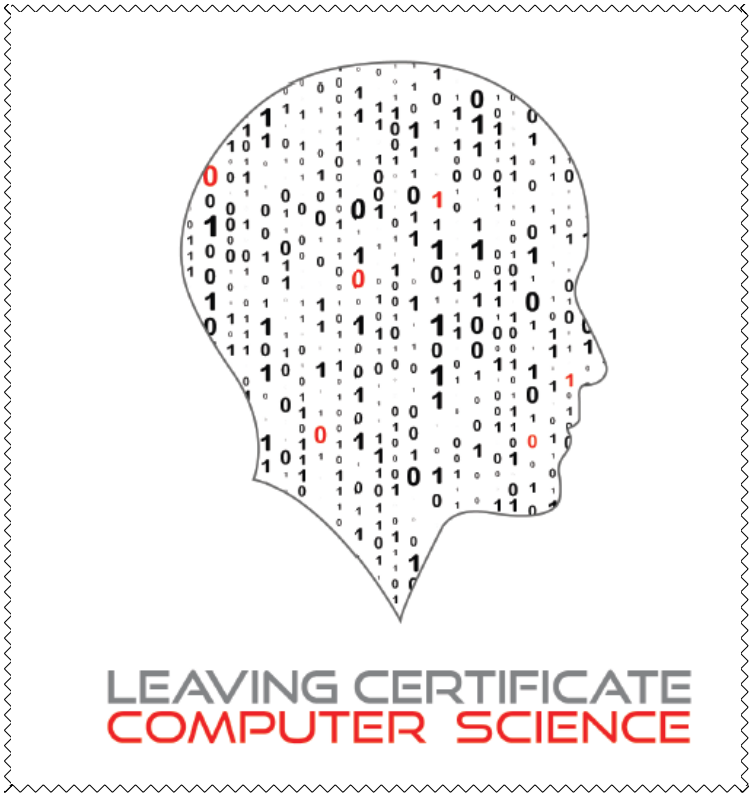
# LC CS Python

## Student Manual




# Section 5

## Programming Logic 1

## *If* statements

**Name: _____**

# Introduction

Thus far, we have been dealing with *sequential* programs i.e. programs which begin their execution at the first line and execute each line in order until the last line is reached.

In addition to sequence, Python supports two other control structures known as *selection* and *iteration*. The purpose of this section is to explain the syntax and semantics of selection and iteration, and explore some common programming techniques used to apply them in real-world contexts.

### Selection

Selection structures are commonly referred to as *decisions*. These structures provided programmers with a branching mechanism whereby, certain blocks of code may be either executed or skipped at runtime. The decision of which block of code to select for execution depends on the result of a *condition* also known as a *Boolean expression*.

The main Python keywords used to support decision structures are `if`, `else` and `elif`.

### Iteration

Iteration structures are commonly referred to as *loops*. Loops are used to cause the same block of code to be executed multiple times. At runtime, the code inside a loop (the *loop body*) is executed repeatedly as long as some condition (the *loop guard*) is met. The loop guard is also a *Boolean expression*.

The main Python keywords used to support iteration structures are `for`, and `while`.
Three other (less important) keywords that relate to loops are `break`, `continue` and `pass`.

**KEY POINT:** Selection and iteration are two programming techniques whose runtime operation is based on the outcome of Boolean expressions

### Boolean Expressions

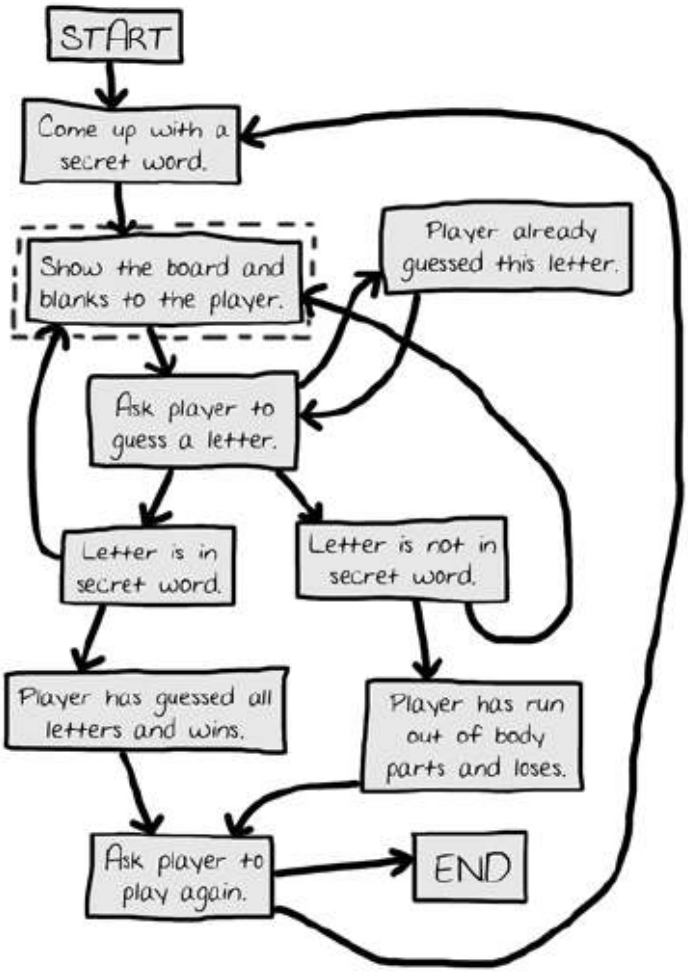
A Boolean expression is any expression that evaluates to either `True` or `False`. They form the basis of all **programming logic**.

# Hangman!

Hangman is a well-known guessing game usually played by two people using pencil and paper. One player thinks of a word and the other tries to guess it by suggesting letters within a certain number of guesses.

- In handout, do **Task 1** (shown below).

The illustration[1] below depicts the main steps of the game and the graphic[2] to the right illustrates a sample run of the game, where the player is trying to guess the word *hangman*.




0 Word: **hangman**
Guess: E
Misses:

1 Word: _ _ _ _ _ _ _
Guess: T
Misses: e

2 Word: _ _ _ _ _ _ _
Guess: A
Misses: e,t

3 Word: _ **A** _ _ _ **A** _
Guess: O
Misses: e,t

4 Word: _ **A** _ _ _ **A** _
Guess: I
Misses: e,o,t

5 Word: _ **A** _ _ _ **A** _
Guess: S
Misses: e,i,o,t

6 Word: _ **A** _ _ _ **A** _
Guess: N
Misses: e,i,o,s,t

7 Word: _ **A N** _ _ **A N**
Guess: R
Misses: e,i,o,s,t

8 Word: _ **A N** _ _ **A N**
Guess:
Misses: e,i,o,r,s,t

[1] http://calab.hanyang.ac.kr/courses/ISD_taesoo/05_Hangman.pdf

[2] https://en.wikipedia.org/wiki/Hangman_(game)

# Boolean Expressions

Boolean Logic was invented by the mathematician George Boole, 1815-1864 who was the first professor of Mathematics at University College Cork (UCC). The algebra on which Boolean logic is based is used extensively to build electronic circuits and write computer programs. Boolean logic, therefore, forms the basis of all modern digital devices and software systems.

Boolean expressions are to Boolean algebra, what algebraic expressions are to algebra, and arithmetic expressions are to arithmetic. At any given moment in time, a Boolean expression will evaluate to either `True` or `False`. It can never be anything in between.

Boolean expressions are so important that it could be argued that the secret to good programming lies in the formation of good Boolean expressions. This is the responsibility of the programmer.

**KEY POINT:** All Boolean expressions evaluate to one of two values - `True` or `False`.

`True` and `False` are two Python keywords which technically behave as if they were the numbers 1 and 0.

**Simple Boolean Expressions**

A *simple Boolean expression* is one that uses a single relational operator (e.g. greater than, less than or equal to etc.) to compare two values.

For example, $7 > 3$ is a simple Boolean expression that compares the numbers 7 and 3 under the relation of 'greater than'. It evaluates to `True` because 7 *is* a bigger number than 3. On the other hand, the expression $7 < 3$ evaluates to `False`, because seven is not less than three.

**STUDENT TIP**
Teachers should provide students with frequent opportunities to extend their technical vocabulary. For example, the terms *Boolean expression* and *condition* can be used interchangeably.

Simple Boolean expressions are the basic building blocks used to implement decisions and loops in Python. Python supports the six relational operators given below.

| Operator | Description | Example | Result |
|---|---|---|---|
| > | Greater than | 7 > 5 | True |
| >= | Greater than or equal to | 7 >= 5 | True |
| < | Less than | 7 < 5 | False |
| <= | Less than or equal to | 7 <= 5 | False |
| == | Equal to (the same as) | 7 == 5 | False |
| != | Not equal to (not the same as) | 7 != 5 | True |

*Python relational operators*

Relational operators are *binary operators* because they need two operands (one either side) in order to work. Although in practice operands are usually numeric, operands can be of any datatype that results from a Python expression. String operands, for example, can be compared using lexicographic ordering of their constituent characters.
Some more examples of simple Boolean expressions (aka conditions) are presented below. (Note: $x = 1$, $y = 0$ and $z = -1$.)

| Condition | Result | Condition | Result |
|---|---|---|---|
| $6 >= 5$ | True | $5 > x$ | True |
| $0 > 1$ | False | $x > y$ | True |
| $1 < 0$ | False | $x <= y$ | False |
| $1 == 0$ | False | $y <= 0$ | True |
| $4 == 4$ | True | $z > y$ | False |
| $4 <= 4$ | True | $x == z$ | False |
| $3 != 4$ | True | $0 == y$ | True |
| $3 <= 4$ | True | $x != y$ | True |

**Compound Boolean Expressions**

Compound Boolean expressions are formed by connecting simple Boolean expressions together using any of the three Python Boolean operators - `and`, `or`, and `not`.

**KEY POINT:**

Boolean operators can only operate on Boolean values i.e. `True/False`.

Just like simple Boolean expressions, compound Boolean expressions always evaluate to either `True` or `False`.

The combinations of values for inputs and their corresponding outputs for `and`, `or`, and `not` can be conveniently represented in a tabular format known as a *truth table*.

`not` is the simplest of the three Python Boolean operators. It is a *unary operator* meaning it only works on one operand at a time. The truth table showing the relationship between some proposition A and `not` A is shown below.

| A | not A |
|---|---|
| False | True |
| True | False |

Both `and` and `or` are *binary operators* meaning that they need two operands to work. The truth tables for `and`, and `or` are shown below.

| A | B | A and B |
|---|---|---|
| False | False | False |
| False | True | False |
| True | False | False |
| True | True | True |

| A | B | A or B |
|---|---|---|
| False | False | False |
| False | True | True |
| True | False | True |
| True | True | True |

The (binary) inputs are given by the columns A and B and the output for these inputs is shown in the rightmost column.

**Examples**

For the purpose of the examples shown below we will assume that we have a number of variables assigned as, $x = 1,\ y = 0,\ z = -1$ and $valid = True,\ finished = False$

| Condition | Result |
|---|---|
| $x == 1$ and $y == 0$ | `True` |
| $x == y$ or $z == -1$ | `True` |
| $x != y$ and $y != -z$ | `True` |
| not valid | `False` |
| $z \leq y$ and finished | `False` |
| $z > y$ or valid | `True` |
| finished and not valid | `False` |
| not finished or not valid | `True` |

- In handout, do **Task 2**.

## The Guessing Game

The remainder of this section will focus on the concepts of selection, iteration and programming logic. A basic 'guess the number' game is used as a platform on which to develop ideas and techniques associated with these concepts. As new concepts are introduced they are exemplified by incorporating them so that they add functionality into the game program. The result is seven versions as follows.

**Guess Game v1:** This is a basic guess game. The base program generates a random number which the user is asked to guess. If the user guess is correct the program displays an appropriate message.

**Guess Game v2:** This time the program displays a message informing the user that they were either correct or incorrect based on the value entered.

**Guess Game v3:** In this version of the game the user is provided with more detailed feedback about their guess i.e. correct, too high or too low.

**Guess Game v4:** This is the same as version 3 except that the user is given at most three chances to guess the correct number. If the user guess correctly within the three allocated chances the program terminates.

**Guess Game v5:** In this version of the game, the program continues until the user makes the correct guess – a subtle but important and powerful enhancement on the previous version. Each time the user enters a guess the program continues to display one of the three messages, i.e. correct, too high or too low.

**Guess Game v6:** A refinement on the previous version whereby after guessing correctly, the user is offered the opportunity to play the game again. If the user enters "N" for no the game exits. Otherwise the program generates a new random number and the game starts again.

**Guess Game v7:** In this final version of the game, the functionality of the game is the exact same as version 6. However, this version validates any data entered by the user i.e. it checks that the guess, is in fact, a number before proceeding.

## Selection (`if`, `else`, `elif`)

Selection statements are used by programmers to build alternative execution paths through their code. As already stated they are commonly referred to as decision statements.

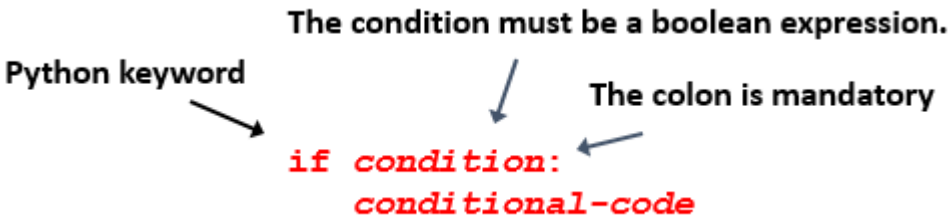
Python provides built in supports for three different kinds of selection statements:

- single option (the basic `if` statement)
- double option (the `if-else` statement)
- multiple option (the **`if-elif-[else]`** statement)

When a running program executes a selection statement, it evaluates a condition, and based on the result of this evaluation it will decide which statement(s) to execute next.

### The basic `if` statement

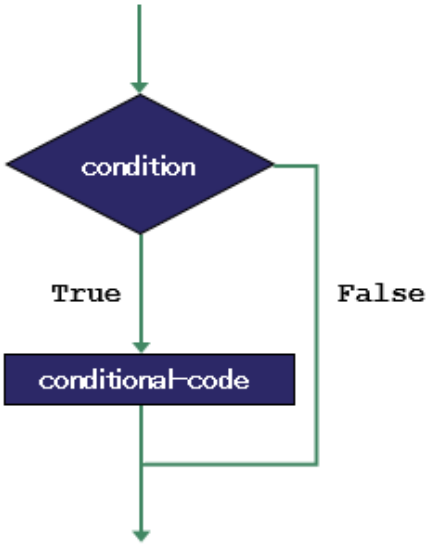
The syntax and semantics of Python's single option if-statement are illustrated and described below.




If Python evaluates the condition to `True`, then the conditional code inside the `if` statement will be executed.

If the condition evaluates to `False`, then the conditional code is skipped and execution continues from the next line of code after the `if`-statement.

Note the use of the colon at the end of the line and also the fact that the *conditional code must be indented*.




*Flow chart illustration of `if`-statement*

**Example (Guess game v1)**

The program below generates a random number between 1 and 10 (`number`) and prompts to user to guess the number (`guess`).

```
1.  # A program to demonstrate the single if statement
2.  import random
3.
4.  number = random.randint(1, 10)
5.  # print(number)
6.
7.  guess = int(input("Enter a number between 1 and 10: "))
8.
9.  # Evaluate the condition
10. if guess == number:
11.     print("Your guess was correct")
12.     print("Well done!")
13.
14. print("Goodbye")
```

*Guessing Game v1*

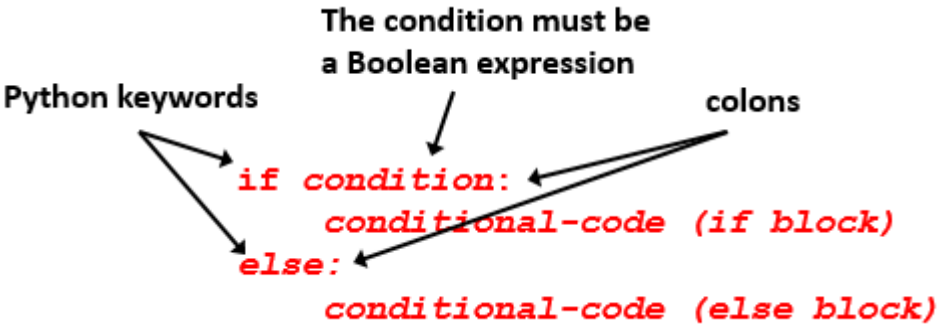
*(Uncommenting line 5 will help you test this program faster)*

- The `if` statement on line 10 evaluates the condition **`guess == number`**
- The execution of lines 11 and 12 are conditional upon the result of this evaluation. Notice the indentation of these lines. They will be executed only if the condition evaluates to `True`
  - The condition will evaluate to `True` if the guess entered by the user is the same as the computer's generated number
  - The condition will evaluate to `False` if the guess entered by the number is *not* the same as the computer's generated number
- The last line is always executed (unconditionally)

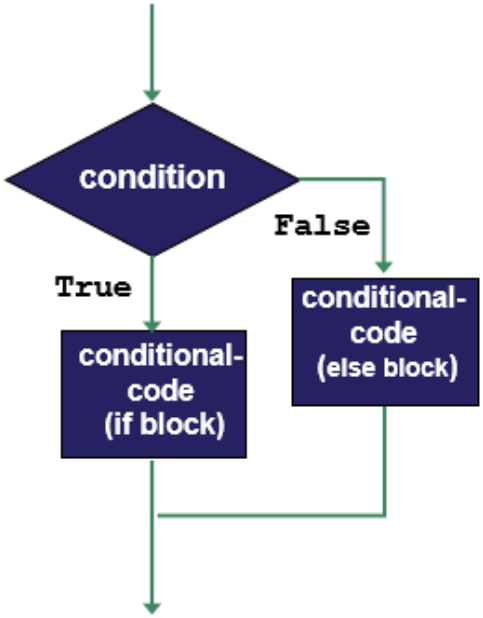
- In handout, do **Task 3** (code shown above)..

### The `if-else` statement

The syntax and semantics of Python's double option if-statement are illustrated and described below.




If Python evaluates the condition to `True`, the block of code associated with the `if`-statement (i.e. the if-block) is executed.
Otherwise, the block of code associated with the `else` statement (i.e. the else-code block) is executed.
In other words, the `else` block is executed only when the condition is evaluated by Python to `False`.
Once either block has been executed the flow of control continues at the next line immediately following the else-block



*Flow chart illustration of `if-else`-statement*

**KEY POINT:**

All conditional code must be indented to the same level by the programmer.

Finally, it is important to recognise that the two blocks ('if' and 'else') are *mutually exclusive*. This means, in any given run of the program either one block or the other will be executed, but never both.

**Example (Guess game v2)**

This example extends guessing game v1 by displaying some messages to the user if they guess the wrong number.

```
1.  # A program to demonstrate the double if statement
2.  import random
3.
4.  number = random.randint(1, 10)
5.  print(number) # comment this line out later!
6.
7.  guess = int(input("Enter a number between 1 and 10: "))
8.
9.  # Evaluate the condition
10. if guess == number:
11.     print("Your guess was correct")
12.     print("Well done!")
13.     print(" ..... play again soon!")
14. else:
15.     print("Hard luck!")
16.     print("Incorrect guess")
17.     print(" ..... play again soon!")
18.
19. print("Goodbye")
```

*Guessing Game v2*

- The condition **`guess == number`** on line 10 is pivotal here again
  - Lines 11 – 13 are selected for execution if the condition evaluates to `True`
  - Lines 15 – 17 are selected for execution if the condition evaluates to `False`

- In handout, do **Task 4** (code shown above)..

**STUDENT TIP**

A common student misconception is that both branches of an `if-else` statement are *always* executed. Teachers should encourage students to use test data that will activate each branch in separate runs of the program.

- Python will always execute the last line of the above program as it is not part of the `if-else` statement

  - In handout, do **Task 5** (code shown above).

***Compare the logic of the two code snippets below. What do you notice?***

```
# Evaluate the condition
if guess != number:
    print("Hard luck!")
    print("Incorrect guess")
else:
    print("Your guess was correct")
    print("Well done!")

print(" ..... play again soon!")
print("Goodbye")
```

```
# Evaluate the condition
if guess == number:
    print("Your guess was correct")
    print("Well done!")
    print(" ..... play again soon!")
else:
    print("Hard luck!")
    print("Incorrect guess")
    print(" ..... play again soon!")

print("Goodbye")
```

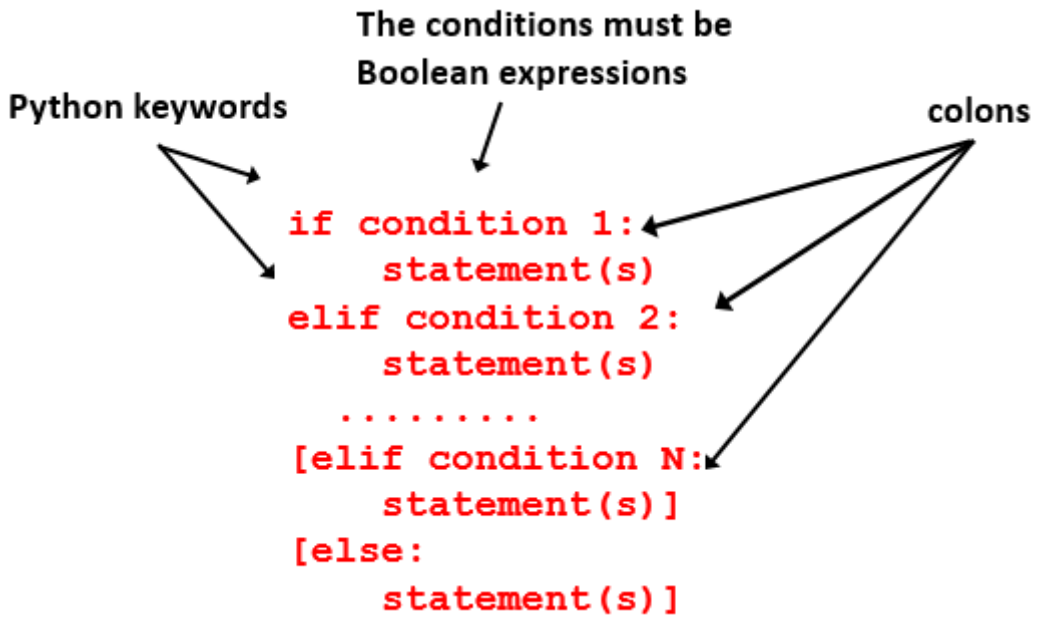
  - In handout, do **Task 6**.

**STUDENT TIP**

It is a very common pitfall to use the single equals (=) assignment oprtator unintentionally instead of the double equals (==) relational operator

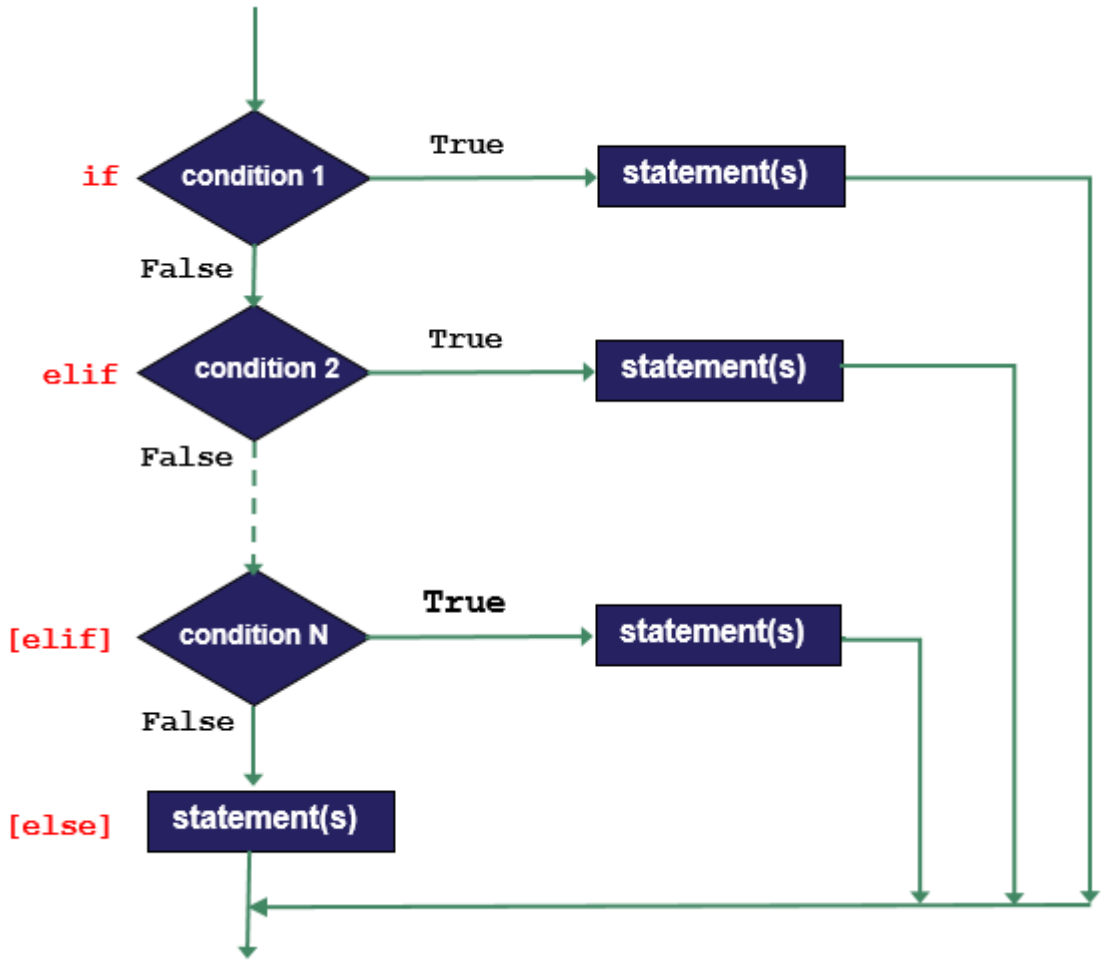
**The `if-elif-[else]` statement**

The syntax and semantics of Python's multiple option if-statement are illustrated and described below.




- The first condition is always inside an `if` statement
- There can be as many `elif` statements as required
- Each `elif` statement must include a condition
- The use of a final `else` statement is optional (indicated by square brackets)
- The `if`, `elif` and `else` keywords must all be at the same level of indentation.
- A colon must be used at the end of any lines containing `if`, `elif` and `else`
- Each condition is evaluated in sequence. Should Python evaluate a condition to `True` then the associated statement(s) are executed and the flow of control continues from the next line following the end of the entire `if-elif` statement. If none of the conditions are found to be `True`, then Python executes any statement(s) associated with the `else`.

The logic of an `if-elif-[else]` statement is illustrated using the flowchart below.




*Flow chart illustration of `if-elif-[else]`-statement*

***Use the space below to reflect on what you have learned about Python's three types of selection statements.***

**Example (Guess game v3))**

This example enhances guessing game v2 by displaying more helpful messages to the user when they make an incorrect guess.

```
1.  # A program to demonstrate the multiple if statement
2.  import random
3.
4.  number = random.randint(1, 10)
5.  print(number) # comment this line out later!
6.
7.  guess = int(input("Enter a number between 1 and 10: "))
8.
9.  # Evaluate the condition
10. if guess == number:
11.       print("Correct")
12.       print("Well done!")
13. elif guess < number:
14.     print("Hard luck!")
15.     print("Too low")
16. else:
17.     print("Hard luck!")
18.     print("Too high")
19.
20. print("Goodbye")
```

*Guessing Game v3*

Think about it – when you ask someone to guess a number between one and ten there are exactly three possible outcomes. The guess can be

- the same as the number you are thinking of
- lower than the number you are thinking of
- higher than the number you are thinking of

**KEY POINT:** The multiple-option if statement should be used to model situations from the real world where there are multiple possible outcomes that require separate specific processing. In this example, that processing is a message tailored to the user's response.

- In any given run of the above example Python will execute either lines 11 and 12, or 14 and 15, or 17 and 18. These lines are mutually exclusive.

- As was the case with the earlier versions of this program Python will always execute the last line of the above program as it is not part of the `if-elif-else` statement

- In handout, do **Task 7** (code shown above).

- In handout, do **Task 8** (shown above).

***Fill in the blanks below without altering the logic of the example program on the previous page. Log your thoughts as you proceed.***

```
if guess > number:
    
elif guess == number:
    
else:
    
```

```
if number     guess:
    print("Hard luck!")
    print("Too low")

else:
    
```