Branching and Looping

These notes provide an overview of the conditional execution of computer code. Two broad categories of condition coding are branching and looping. Branching is the conditional selection of the code that will be executed. I.e., branching changes the direction of a program's flow of execution based on whether a specified condition is True or False. A loop is a sequence of instructions that is repeated until a specified condition is met. A loop performs an operation, then checks the specified condition to determine whether the loop is be repeated or whether it has been completed.

Conditional Execution of Code

Conditional statements are logical expressions (e.g., if…, then…) that direct the flow of executed code in a computer program. When encountered, a conditional statement is evaluated, yielding a logical value of True or False. If the condition is true, then a particular block of code is executed; if the conditional statement is false, the "conditional block of code" is bypassed (an alternative block of code may be executed instead).

In Python, conditional statements include if, if else, for loops, and while loops.

Passwords, something we regularly encounter, are "if…, then…" logical expressions. A user is asked for a password, which is evaluated as correct or incorrect. *If* the user entered the correct password (True), *then* they can access the computer/software/website/etc. If the password is incorrect (False), the user is denied access.

Python supports the standards logical conditions from mathematics:

Equal: a == b

Not Equal: a != b

Less than: a < b

Less than or equal to: a <= b

Greater than: a > b

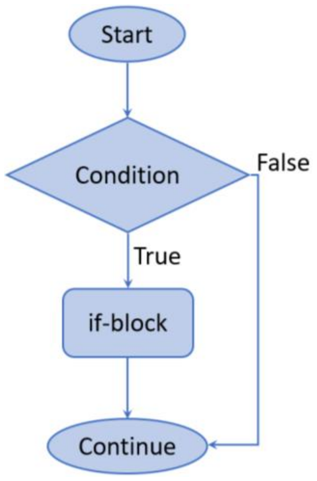
Greater than or equal to: a >= b

These conditions are commonly in if statements and loops.

<u>if Statements</u>

An if statement is used when you want to execute a block of code when a specified condition is True.

<u>Flow of an if statement</u>:



First, the condition (an if statement in this case) is checked

If the condition evaluates to True, the statements in the if-block are executed

If the condition evaluates to False, the if-block is bypassed

<u>Simple example</u>:

```
a = 76
b = 238
if b > a:
  print('b is greater than a')
Output:
b is greater than a
```

<u>Note</u>: The colon at the end of the if statement and indentation are very important – Python relies on them to determine the if-block of code.

<u>Shorthand if statement</u>: When there is only one statement to execute when a condition is evaluated as True, then the statement can appear on the same line as the if statement.

```
# One line if statement
if a > b: print('a is greater than b')
Output:
a is greater than b
```

and

The 'and' keyword is a logical operator used to combine conditional statements; e.g.,

```
a = 160

b = 33

c = 345

if a > b and c > a:

  print('Both conditions are True')

Output:

Both conditions are True
```

or

The 'or' keyword is also logical operator; it is used to combine conditional statements; e.g.,

```
a = 182

b = 14

c = 667

if a > b or a > c:

  print("At least one of the conditions is True")

Output:

At least one of the conditions is True
```

Nested if statements

A nested if statement occurs when there are if statements inside of if statements; e.g.,

```
x = 10

y = 5

if x < y:

    print('x is less than y')

    if x >= y:

        print('x is greater than or equal to y')

Output:

x is greater than or equal to y
```

elif

The 'elif' statement allows the evaluation of multiple logical expressions. Conditions are evaluated until one of them is evaluated as TRUE, then the block of code associated with that condition is executed. Any additional conditions are ignored after a TRUE condition is encountered.

```
a = 33

b = 33

if b > a:

  print('b is greater than a')

elif a == b:

  print('a and b are equal')

Output:

a and b are equal
```

Shorthand if-else: If there is only one statement to execute for if and only one to execute for else, the conditional statements and their associated code can appear on the same line:

```
a = 2

b = 330

print('A') if a > b else print('B')

Output:

B
```

Multiple else statements can appear on the same line of code. Here is a shorthand if-else statement with 3 conditions:

```
a = 330

b = 330

print("A") if a > b else print("=") if a == b else print("B")

Output:

=
```

else

The 'else' keyword indicates how to proceed if no earlier conditions were evaluated as True.

```
a = 200

b = 33

if b > a:

  print('b is greater than a')

elif a == b:

  print('a and b are equal')

else:

  print('a is greater than b')

Output:

a is greater than b
```

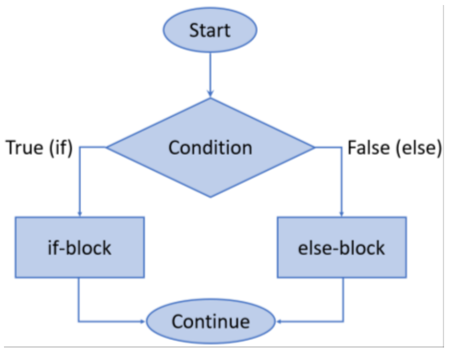
if-else Statements

Note: 'else' can be used without 'elif'.

if-else statements are used when there are two possible outcomes, each with its own set of code to be executed. When the logical expression is True ('if'), one block of code is executed; then the condition is False ('else'), a different block of code is executed.

General syntax:

```
if <logical expression>:   # I.e., conditional statement

   if-block of code        # Executed if the condition evaluates to True

else:

    else-block of code     # Executed if the condition evaluates to False
```



First, the condition is evaluated

If the condition evaluates to True, the statements in the if-block are executed

If the condition evaluates to False, the else-block is executed

Simple example:

```
a = 200
b = 33
if b > a:
  print('b is greater than a')
else:
  print('b is not greater than a')
Output:
b is not greater than a
```

Another example:

```
cntry = 'Bolivia'
if cntry == 'US':
   print('Capital is Washington DC')
else:
   print('Bolivia has two capitals - Sucre and La Paz')
Output:
Bolivia has two capitals - Sucre and La Paz
```

Chained Conditionals

If a logical expression has more than two possibilities, then the conditional code will have more than two branches. One way to code this is with a chained conditional:

```
if x < y:
    Code_A
elif x > y:
    Code_B
else:
    Code_C
```

Encountering the `'if'` statement, Python evaluates it. If the `'if'` condition is True, then Code_A is executed.

If the `'if'` condition is not True, then the `'elif'` condition is evaluated; if it is True, then Code_B is executed.

If the neither the `'if'` nor `'elif'` conditions are True, then the `'else'` condition is True and Code_C is executed.

Note: Conditions are checked in order. Only the code associated with the first condition that evaluates as True is executed – even if more than one condition is True, only the code associated with the first True condition executes.

Note: There can be multiple `'elif'` statements in a block of code.

Loops

Loops are essential building blocks for most computer algorithms. They are used to repeatedly execute a block of code a desired number of times. This type of code flow is called a `'loop'` because it loops back around to the top of the block after each iteration. In Python, `'for loops'` and `'while loops'` are commonly used.

for loops

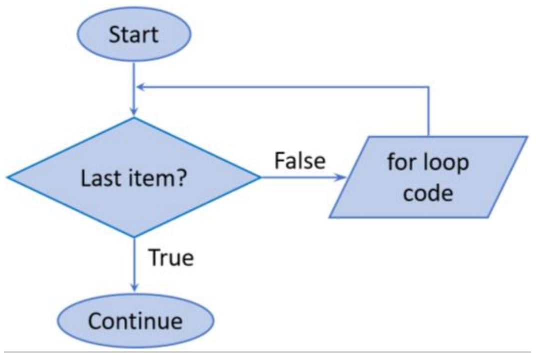
A for loop is used to sequentially iterate over an iterable object such as a string, tuple, list, etc., performing the same action for each entry.

To create a for loop, first define the iterable object you'd like to loop through. Then specify the actions to be performed on each item in that iterable object.

Basic syntax:

```
for variable_name in sequence:
     <block of code >
```

Flow of a for loop:



Program enters the loop and checks the last item in the sequence has been reached

If so, loop is exited; if not, for loop code is executed

When the for loop code finishes executing, the code checks whether the last item in the sequence has been reached

If so, loop is exited; if not, for loop code is executed again

Etc.

When the last item is reached, the loop is exited

Simple example:

```
SEC_list = ['Razorbacks', 'Bulldogs', 'Aggies', 'Tigers']

for name in SEC_list:

    print(name)

Output:

Razorback

Bulldogs

Aggies

Tigers
```

Note that in the example above, the variable `'name'` was not defined. There's no need to define it because for loops iterate through objects in sequence, so this variable can actually be called almost anything. Python interprets any variable following `'for'` as referring to each list entry in sequence as the loop executes. I.e., `'name'` points to `'Razorbacks'` on the first iteration, then `'name'` points to `'Bulldogs'` on the second iteration, and so on.

The above example was simple, but for loops are powerful – complex instructions can be nested inside of them.

Loops typically used to perform calculations. For example, a for loop can be used to calculate the average range of electric vehicles (EVs) stored in a list. To do this, sum the ranges of all of the EVs and then divide them by the total number EVs in the list.

A more advanced example:

```
total_range = 0              # Create variable to store total range value

for row in ev_data[1:]:      # Loop over rows, starting w/ row 2 (index 1)

    ev_range = row[1]        # A car's range is found in col 2 (index 1)

    total_range += ev_range  # Add next value to number in total_range

number_of_cars = len(ev_data[1:])   # Calc list length, minus header row

print(total_range/number_of_cars)   # Print the average range

Output:

224.0
```

for loop with else

A for loop can have an optional else block. The else block is executed when the items in the sequence used in the for loop are exhausted.

Example of an else block in a for loop:

```
digits = [0, 1, 5]

for i in digits:

   print(i)

else:

   print('No items left.')
```

Output:

```
0

1

5

No items left.
```

The for loop iteratively prints the items in the list until they are exhausted; when the for loop "exhausts," the else block of code will execute, resulting in the printing of 'No items left' in this case.

Break statements

The keyword 'break' can be used to stop a for loop. In this case, the else part of the for loop is ignored. Thus, a for loop's else part runs if no break occurs. A break statement stops the execution of the loop before it has looped through all the items in a sequence:

Example: Exit the loop when x is "banana"

```
fruits = ['apple', 'banana', 'cherry']
for x in fruits:
  print(x)
  if x == 'banana':
    break
Output:
apple
banana
```

Note: The loop processes for the `'break'` term because `'print(x)'` comes before `'break'`, but the loop does not continue after the `'break' statement`.

Example: Exit the loop when x is `'banana'`, but this time the break comes before `'print()'`

```
fruits = ['apple', 'banana', 'cherry']
for x in fruits:
  if x == 'banana':
    break
  print(x)
Output:
apple
```

Continue statement

A continue statement stops the current iteration of the loop, then continues with the next iteration

Example: Do not print banana

```
fruits = ['apple', 'banana', 'cherry']
for x in fruits:
  if x == 'banana':
    continue
  print(x)
```

```
Output:
apple
cherry
```

Nested for loop

A nested loop is a loop inside a loop. The `'inner loop'` will be executed one time for each iteration of the `'outer loop'`.

Example: Print each adjective for every fruit

```
adj = ['red', 'big', 'tasty']

fruits = ['apple', 'banana', 'cherry']

for x in adj:

  for y in fruits:

    print(x, y)
```

```
Output:

red apple

red banana

red cherry

big apple

big banana

big cherry

tasty apple

tasty banana

tasty cherry
```

Iteration	**x**	**y**
1	red	apple
2	red	banana
3	red	cherry
4	big	apple
5	big	banana
6	big	cherry
7	tasty	apple
8	tasty	banana
9	tasty	cherry

while loops

A while loop repeatedly executes a block of code *as long as* (i.e., while) a condition is True. A while loop is essentially a repeating conditional statement — after an if statement, the program continues to execute code within the while loop. When finished executing the code, the program jumps back to the start of the while statement and repeats the execution of the code until the condition is False.

Note: A while loop requires the creation of a counter (index) variable prior to entry into the loop.

Basic syntax:

```
while <condition>:      # True – execute loop; False – exit loop

      <loop code>       # body of while loop
```

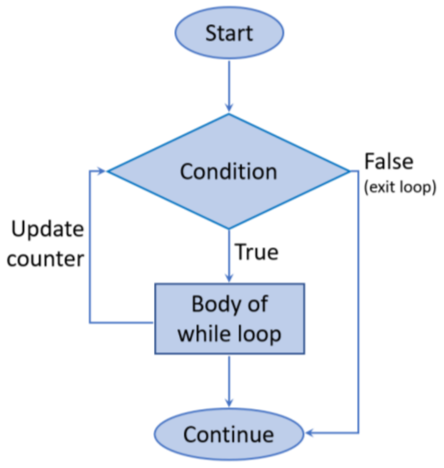
Simple example:

```
i = 1

while i < 6:

  print(i)

  i += 1
```

Note: The counter, i, keeps track of the number of iterations. It must be incremented, otherwise the loop would continue forever.

Note: In Python, the body of a while loop is determined through indentation. The body of the loop starts with indentation and the first unindented line marks the end of the loop.

Flow of while loop:



Set index = 1

Enter loop and check condition, index <= n

If the condition is True, execute loop code

    After executing code, update index by 1

    Check condition – if True, execute code; if False, exit

    Repeat until condition is False

If condition is False, exit loop

Example of a while loop:

```
# Code to add natural numbers up to sum = 1 + 2 + 3 + ... + n

n = 10  # Any value of n can be chosen

# Initialize 'sum' [start sum at zero] and 'counter' [set at 1 to start
the count]

sum = 0

i = 1

# while loop

while i <= n:

    sum = sum + i

    i = i + 1     # Update 'counter'

# Print the sum

print('The sum is', sum)

Output:

The sum is 55
```

In the above code, the condition (`'i <= n'`) is True *as long as* the counter variable `'i'` is less than or equal to n (10 in the code above).

The counter variable is initialized at `'i = 1'`, but the value of `'counter'` must be increased each time the body of the while loop is executed. This is very important and should not be forgotten – failing to do so will result in an infinite loop (one that iterates forever [or until you kill the program).

while loop with else

This is similar to the while loop above, except that an else block of coded is included in the body of the loop. The else part of the loop code is executed if the condition in the while loop evaluates to False.

The while loop can be terminated with a break statement. In such cases, the else part is ignored. Hence, a while loop's else part runs if the condition is False and no break occurs.

Example:

```
counter = 0
while counter < 3:
    print('Inside loop')
    counter = counter + 1
else:
    print('Inside else')
```

Output of the above code:

```
Inside loop
Inside loop
Inside loop
Inside else
```

In the example above, the counter variable is used to have the string 'Inside loop' printed three times (when counter = 0, 1, 2). On the fourth iteration, the condition in while becomes False (counter = 3, so counter < 3 is False). When the condition is False, the else part of the while body is executed.

Break statements

A break statement ends the loop even if the while condition is still True; e.g.,

```
i = 1
while i < 6:
  print(i)
  if (i == 3):
    break
  i += 1
```

Output:

```
1
2
3
```

Note: “3” is printed because ' break' appears after 'print(i)'.

Continue statements

The statement `'continue'` stops the current iteration and continues with the next iteration; e.g.:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

Output:

```
1
2
4
5
6
```

Note: “3” is missing in the output because the 3rd iteration was stopped by `'continue'`, then looping resumed with i = 4.

while loop with an else statement

With an else statement, a block of code associated with `'else'` can be run once the condition no longer is True:

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print('i is no longer less than 6')
```

Output:

```
1
2
3
4
5
i is no longer less than 6
```

range() function

A specified sequence of numbers can be generated using Python's range() function. For example, range(10) will generate the numbers 0 to 9 (note: 'range(10)' returns 0 to 9, not 1 to 10). The sequence of numbers generated by 'range()' can be used to loop through a block of code a specified number of times.

The start, stop, and step size of the range can be specified as 'range(start, stop, step_size)'. If not specified, step_size defaults to 1.

Examples of using range():

```
print(range(10))

print(list(range(10)))

print(list(range(2, 8)))

print(list(range(2, 20, 3)))

Output:

range(0, 10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[2, 3, 4, 5, 6, 7]

[2, 5, 8, 11, 14, 17]
```

The range() function can be used in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate through a sequence using indexing. Here is an example:

```
# Code to iterate through a list using indexing

genre = ['pop', 'rock', 'jazz']

# Iterate over the list using index

for i in range(len(genre)):

    print("I like ", genre[i])

Output:

I like pop

I like rock

I like jazz
```