Iterations

Last updated on 2024-05-23 | Edit this page

Download Chapter notebook (ipynb)

Mandatory Lesson Feedback Survey

OVERVIEW

Questions

- What do we mean by *iterations* and *loops*?
- How are `for`-loops implemented?
- Can conditional statements be used in iterations?
- When to use `while`-loops?

Objectives

- Understanding the concept of iterations and loops.
- Learning the processes involved in `for`-loops implementation.
- Building concept of using conditional statements in loops.
- Understanding when to use while loop.

This chapter assumes that you are familiar with the following concepts in Python 3:

PREREQUISITE

- I/O Operations
- Variables and Types
- Mathematical Operation
- Logical Operations
- Indentation Rule
- Conditional Statements
- Arrays

Additionally, make sure that you are comfortable with the principles of indexing in arrays before you start this section. It is very important that you have a good understanding of arrays and sequences, because the concept of iteration in programming deals *almost* exclusively with these subjects.

NOTE

You can practice everything in this section and the subsequent ones as you have been doing so far. However, if you find it hard to grasp some of the concepts, don't worry, you are not alone. It takes practice. To help you with that, Philip Guo from UC San Diego (Calif., USA) has developed PythonTutor.com, an excellent online tool for learning Python. On that website, write (or 'copy and paste') your code in the editor, then click *Visualize Execution*. In the new page, use the *forward* and *back* buttons to see a step-by-step graphical visualisation of the processes that occur during the execution of your code. Try it on the examples in this section.

The concept

We employ iterations and loops in programming to perform repetitive operations. A repetitive operation is a reference to one or several defined operations that are repeated multiple times.

For instance, suppose we have a `list` of 5 numbers as follows:

```
numbers = [-3, -2, 5, 0, 12]
```

Now we would like to multiply each number by 2. Based on what we have learned thus far, this is how we would approach this problem:

PYTHON

```
numbers = [-3, -2, 5, 0, 12]

numbers[0] *= 2
numbers[1] *= 2
numbers[2] *= 2
numbers[3] *= 2
numbers[4] *= 2

print(numbers)
```

OUTPUT

```
[-6, -4, 10, 0, 24]
```

Whilst this does the job, it is clearly very tedious and repetitive. In addition to that, if we have an array of several thousand members, this approach becomes infeasible.

The process of multiplying individual members of our array with 2 is a very simple example of a repetitive operations.

REMEMBER

In programming, there is a universally appreciated golden principle known as the **DRY** rule; and this is what it stand for:

Don't **R**epeat **Y**ourself

So if you find yourself doing something again and again, it is fair to assume that there might a better way of getting the results you're looking for ...

Some programmers (with questionable motives) have come up with a **WET** rule too. Find out more about DRY and WET from Wikipedia.

There are some universal tools for iterations that exist in all programming languages — *e.g.* `for` and `while` loops. Some other tools such as vectorisation or generators, however, are unique to one or several specific programming languages.

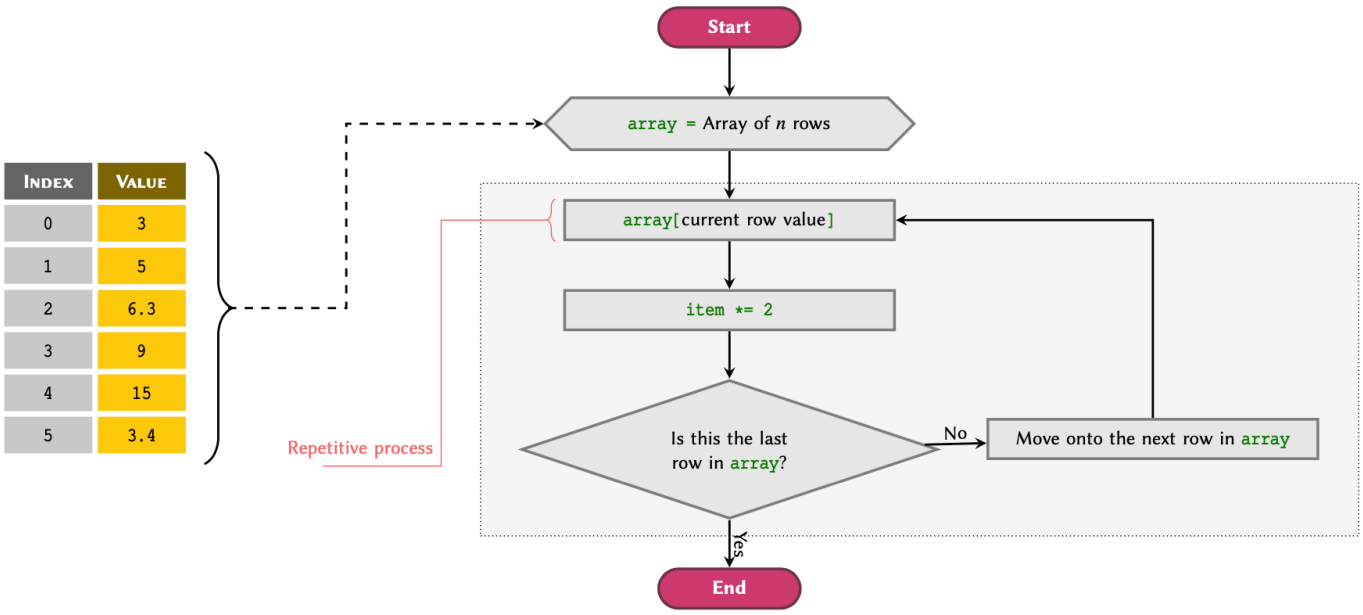
Throughout this section, we will discuss iterations via `for` and `while` loops, and review some real-world examples that may only be addressed using iterative processes.

for-loops

Some of the data show that up to 80% of all conventional iterations are implemented as `for` loops. Whether or not it is the best choice in of all these cases is a matter of opinion. What is important, however, is to learn the difference between the 2 methods and feel comfortable with how they work.

Implementation of `for` loops in Python is simple compared to other programming languages. It essentially iterates through an existing *iterable* variable — e.g. an array, and retrieves the values from it one by one, from the beginning right down to the end.

> REMEMBER
>
> In Python, *iterable* is a term used in reference to variables that can be iterated through. Any variable type that can be used in a `for` loop *without* any modifications is therefore considered an *iterable*.
>
> Most arrays and sequences are iterable. See Table to find out which native types in Python are iterable. A rule of thumb is that if an array or a sequence is numerically indexed (e.g. `list`, `tuple`, or `str`), it is an iterable.



Flowchart of a `for`–loop workflow applied to a `list` array.

Figure illustrates a flowchart to visualise the workflow of an iterative process using `for` loops in Python. The process depicted in the flowchart may be described as follows:

ADVANCED TOPIC

An iterable is a Python variable that contains the built–in method `.__iter__()`. Methods starting and ending with two underscores (dunderscores) are also known as *magic methods* in Python. See Python documentations for additional information.

PROCESS

1. A `for`-loop is initialised using an array or a sequence and begins its process by going through the array values from the first row.
2. **Iterative Process:** The value of the *current row* is retrieved and given the alias item, which now represents a variable in the context of the loop.
3. **Repetitive Operation(s):** Designated operations are performed using the value of item:
    - item *= 2
4. **Loop Condition:** The `for` loop *automatically* checks whether or not it has reached the last row of the sequence:
    - **NO:** Move onto the next row and *repeat* the process from #2.
    - **YES:** Exit the loop.

We write this process in Python as:

PYTHON

```python
numbers = [3, 5, 6.3, 9, 15, 3.4]

for item in numbers:
    item *= 2
    # Display the item to see the results:
    print(item)
```

OUTPUT

```
6
10
12.6
18
30
6.8
```

where we can see that the result for each iteration is displayed in a new line. Example outlines other such applications and expands on repetitive operations that may be simplified using `for` loops.

REMEMBER

A `for` loop is always initialised as:

```
for variable_name in an_array:
  # An indented block of processes
  # we would like to perform on the
  # members of our array one by one.
```

where `an_array` is an *iterable* variable, and `variable_name` is the name of the variable we temporarily assign to a member of `an_array` that corresponds with the current loop cycle (iteration). The number of loop cycles performed by a `for` loop is equal to the length (number of members) of the array that we are iterating through, which in this case is called `an_array`.

You can think of each iteration cycle as pulling out a row from table that is our array (as exemplified in section arrays) and temporarily assigning its corresponding value to a variable until the next iteration cycle.

See subsection List Members to find the length of an array.

DO IT YOURSELF

Given:

PYTHON

```
peptides = [
    'GYSAR',
    'HILNEKRILQAID',
    'DNSYLY'
               ]
```

Write a `for` loop to display each item in peptides alongside its index and length. Display the results in the following format:

`Peptide XXXX at index X contains X amino acids.`

Solution

PYTHON

```python
for sequence in peptides:
    length = len(sequence)
    index = peptides.index(sequence)

    print('Peptide', sequence, 'at index', index, 'contains', length, 'amino acids.')
```

OUTPUT

```
Peptide GYSAR at index 0 contains 5 amino acids.
Peptide HILNEKRILQAID at index 1 contains 13 amino acids.
Peptide DNSYLY at index 2 contains 6 amino acids.
```

EXTENDED EXAMPLE OF ITERATIONS USING `for` LOOPS

When using a `for` loop, we can also reference other variables that have already been defined outside of the loop block:

PYTHON

```
numbers = [3, 5, 6.3, 9, 15, 3.4]
counter = 0

for item in numbers:
    item *= 2

    # Display the item to see the results:
    print('Iteration number', counter, ':', item)

    counter += 1
```

OUTPUT

```
Iteration number 0 : 6
Iteration number 1 : 10
Iteration number 2 : 12.6
Iteration number 3 : 18
Iteration number 4 : 30
Iteration number 5 : 6.8
```

It is also possible to define new variables inside the loop, but remember that the value of any variables defined inside a loop is reset in each iteration cycle:

PYTHON

```
numbers = [3, 5, 6.3, 9, 15, 3.4]
counter = 0

for item in numbers:
    new_value = item * 2

    # Display the item to see the results:
    print('Iteration number', counter, ':', item, '* 2 =', new_value)

    counter += 1
```

OUTPUT

```
Iteration number 0 : 3 * 2 = 6
Iteration number 1 : 5 * 2 = 10
Iteration number 2 : 6.3 * 2 = 12.6
Iteration number 3 : 9 * 2 = 18
Iteration number 4 : 15 * 2 = 30
Iteration number 5 : 3.4 * 2 = 6.8
```

DO IT YOURSELF

Write a `for` loop to display the values of a `tuple` defined as:

PYTHON

```
protein_kinases = ('PKA', 'PKC', 'MPAK', 'GSK3', 'CK1')
```

such that each protein is displayed on a new line and follows the phrase `Protein Kinase X:` as in

```
Protein Kinase 1: PKA
Protein Kinase 2: PKC
```

and so on.

Solution

PYTHON

```
counter = 1

for protein in protein_kinases:
    print('Protein Kinase ', counter, ': ', protein, sep='')
    counter += 1
```

OUTPUT

```
Protein Kinase 1: PKA
Protein Kinase 2: PKC
Protein Kinase 3: MPAK
Protein Kinase 4: GSK3
Protein Kinase 5: CK1
```

Retaining the new values

It is nice to be able to manipulate and display the values of an array but in the majority of cases, we need to retain the new values and use them later.

In such cases, we have two options:

- Create a new array to store our values.
- Replace the existing values with the new ones by overwriting them in the same array.

Creating a new array to store our values is very easy. All we need to do is to create a new `list` and add values to it in every iteration. In other words, We start off by creating an empty `list`; to which we then add members using the `.append()` method inside our `for` loop. The process of creating a new `list` and using the `.append()` method to values to an existing `list` are discussed in subsections Useful Methods and mutability, respectively.

PYTHON

```
numbers = [-4, 0, 0.3, 5]
new_numbers = list()

for value in numbers:
    squared = value ** 2
    new_numbers.append(squared)

print('numbers:', numbers)
```

OUTPUT

```
numbers: [-4, 0, 0.3, 5]
```

PYTHON

```
print('new_numbers:', new_numbers)
```

OUTPUT

```
new_numbers: [16, 0, 0.09, 25]
```

DO IT YOURSELF

Given:

PYTHON

```
peptides = [
    'GYSAR',
    'HILNEKRILQAID',
    'DNSYLY'
               ]
```

write a `for` loop in which you determine the length of each sequence in peptides, and then store the results as a `list` of `tuple` items as follows:

```
[('SEQUENCE_1', X), ('SEQUENCE_2', X), ...]
```

Solution

PYTHON

```
peptides_with_length = list()

for sequence in peptides:
    length = len(sequence)
    item = (sequence, length)

    peptides_with_length.append(item)
```

The replacement method uses a slightly different approach. Essentially what we try to achieve is:

- read the value of an item in an array;
- manipulate the value via operations;
- put the value back to the original array through *item assignment* and thereby replace the existing value.

We learned about modifying an existing value in a `list` in subsection mutability, where we discussed the concept of *item assignment*. The process of replacing the original values of an array in a `for` loop is identical. The key to performing this process, however, is that we need to have the correct *index* for the specific member of the array that we are trying to modify. Additionally, don't forget that *item assignment* is only possible in mutable arrays such as `list`. See Table to see which types of array are mutable in Python.

To perform *item assignment*, we can implement a variable to represent the current iteration cycle in our `for` loop. We do so by initialising the variable with a value of 0, and adding 1 to its value at the end of each cycle. We can then use that variable as an *index* in each iteration cycle:

PYTHON

```
numbers = [-4, 0, 0.5, 5]

# Variable representing the
# index (iteration cycle):
index = 0

for value in numbers:
    new_value = value ** 5

    # Putting it back into
    # the original array:
    numbers[index] = new_value

    # Adding one to the index for
    # the next iteration cycle:
    index += 1

print(numbers)
```

OUTPUT

```
[-1024, 0, 0.03125, 3125]
```

ADVANCED TOPIC

The `enumerate()` function actually returns a *generator* of `tuple` items each time it is called in the context of a `for` loop. A generator is in principle very similar to a normal array; however, unlike an array, the values of a generator are not evaluated by the computer until the exact time at which they are going to be used. This is an important technique in functional programming known as *lazy evaluation*. It is primarily utilised to reduce the workload on the computer (both the processor and the memory) by preventing the execution of processes that may be delayed for a later time. In the case of the `enumerate()` function, the values are evaluated at the beginning of each iteration cycle in a `for` loop. Learn more about lazy evaluation in Wikipedia or read more on generators in Python in the official documentations.

This is a perfectly valid approach and it is used in many programming languages. However, Python makes this process even easier by introducing the function `enumerate()`. We often use this function at the initiation of a `for` loop. The function takes an array as an input and as the name suggests, enumerates them; thereby simplifying the indexing process. The previous example may therefore be written more concisely in Python as follows:

PYTHON

```python
numbers = [-4, 0, 0.5, 5]

for index, value in enumerate(numbers):
    # Manipulating the value:
    new_value = value ** 5
    numbers[index] = new_value

print(numbers)
```

OUTPUT

```
[-1024, 0, 0.03125, 3125]
```

DO IT YOURSELF

Given:

PYTHON

```python
characters = ['1', '2', '3', '4']
```

Display each item in characters as many times in one line as the index of that item in characters. The results should appear as follows:

```
2
33
444
```

Solution

PYTHON

```
for index, item in enumerate(characters):
    print(item * index)
```

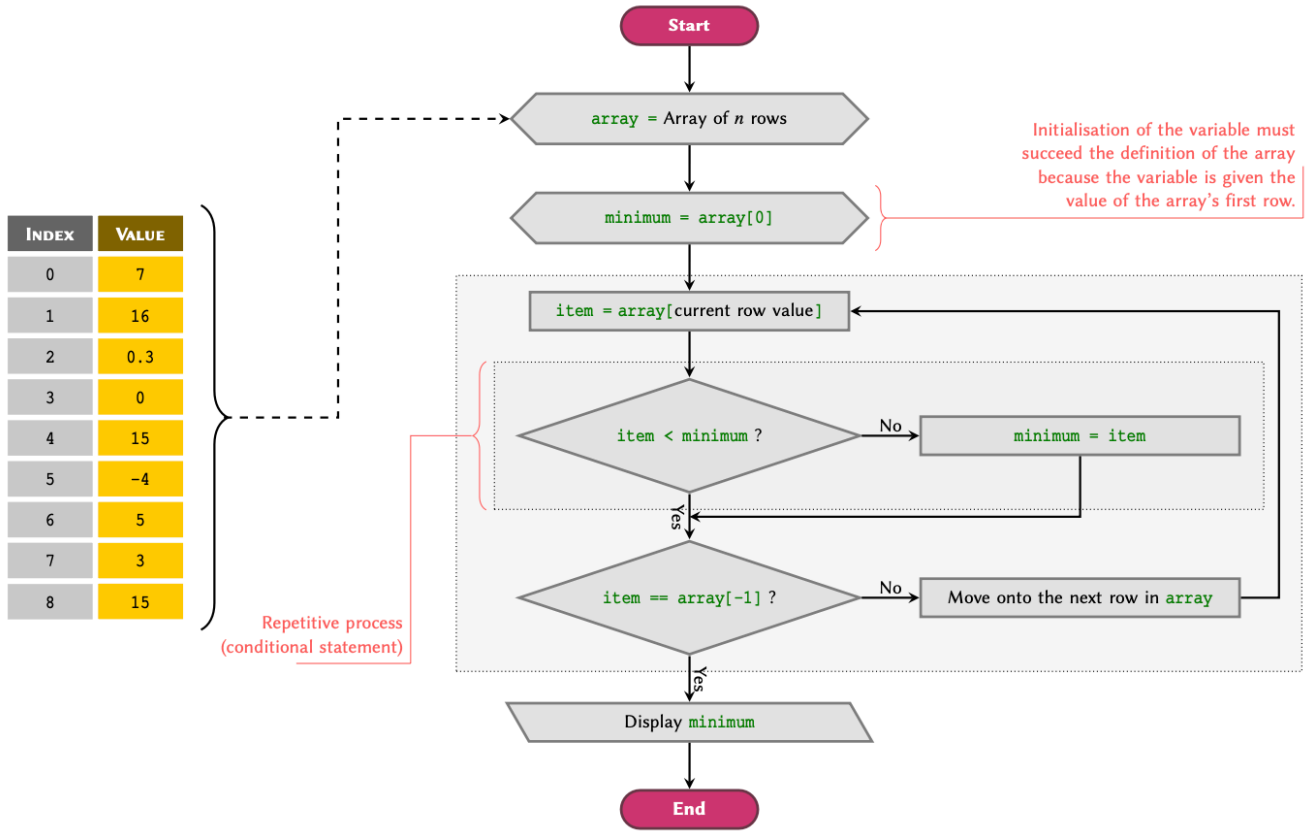
OUTPUT

```
2
33
444
```

for-loop and conditional statements

We can use conditional statements within `for` loops to account for and handle different situations.

Suppose we want to find the smallest value (the minimum) within a `list` of numbers using a `for` loop. The workflow of this process is displayed as a flowchart diagram in figure below.



Given an array, we can break down the problem as follows:

1. Define the variable `minimum` outside our `for`-loop, and set its value to be equal to the value of the first row in our array.
2. A `for`-loop is initialised using an array or a sequence and begins its process by going through the array values from the first row.
3. **Iterative Process:** The value of the *current row* is retrieved and given the alias `item`, which now represents a variable in the context of the loop.
4. **Repetitive Operation(s):** Designated operations are performed using the value of `item`:
    - Compare the value of `minimum` with the value of every row in the array at each iteration cycle as test if `item < minimum`:
        - `True`: update the value of `minimum` to be `minimum = item`.
        - `False`: do nothing and proceed to the next step.
5. **Loop Condition:** The `for`-loop *automatically* checks whether or not it has reached the last row of the sequence:
    - **NO:** Move onto the next row and *repeat* the process from #3.
    - **YES:** Exit the loop.

Finally, we can implement the process displayed in figure as follows:

PYTHON

```
numbers = [7, 16, 0.3, 0, 15, -4, 5, 3, 15]

minimum = numbers[0]

for value in numbers:
    if value < minimum:
        minimum = value

print('The minimum value is:', minimum)
```

OUTPUT

```
The minimum value is: -4
```

DO IT YOURSELF

Given:

PYTHON

```
peptides = [
  'FAEKE',
  'CDYSK',
  'ATAMGNAPAKKDT',
  'YSFQW',
  'KRFGNLR',
  'EKKVEAPF',
  'GMGSFGRVML',
  'YSFQMGSFGRW',
  'YSFQMGSFGRW'
               ]
```

Using a `for` loop and a conditional statement, find and display the sequences in peptides that contain the amino acid serine (S) in the following format:

```
Found S in XXXXX.
```

Solution

PYTHON

```
target = 'S'

for sequence in peptides:
    if target in sequence:
        print('Found', target, 'in', sequence)
```

OUTPUT

```
Found S in CDYSK
Found S in YSFQW
Found S in GMGSFGRVML
Found S in YSFQMGSFGRW
Found S in YSFQMGSFGRW
```

Sequence of numbers in for-loops

To produce a sequence of `int` numbers to use in a `for` loop, we can use the built-in `range()` function. The function takes in 3 positional arguments representing *start*, *stop*, and *step*. Note that `range()` is *only* capable of producing a sequence of integer numbers.

REMEMBER

The `range()` function does not create the sequence of numbers immediately. Rather, it behaves in a similar way as the `enumerate()` function does (as a generator).

Displaying the output of a `range()` function is not, as one might expect, an array of numbers:

PYTHON

```
range_generator = range(0, 10, 2)

print(range_generator)
```

OUTPUT

```
range(0, 10, 2)
```

It is, however, possible to evaluate the values outside of a `for` loop. To do so, we need to convert the output of the function to `list` or a `tuple`:

PYTHON

```
range_sequence = list(range_generator)

print(range_sequence)
```

OUTPUT

```
[0, 2, 4, 6, 8]
```

REMEMBER

The `range()` function is *non-inclusive*. That is, it creates a sequence that starts from and includes the value given as the *start* argument, up to but excluding the the value of the *end* argument. For instance, `range`(1, 5, 1) creates a sequence starting from 1, which is then incremented 1 step at a time right up to 5, resulting in a sequence that includes the following numbers: 1, 2, 3, 4

EXAMPLE: SEQUENCE COMPARISON. DOT PLOTS AND `for`-LOOPS

There are different methods to evaluate the similarity of two or more polypeptide or polynucleotide sequences. The *dot matrix* (also known as the *dot plot*) is one such method. A *dot matrix* is in essence a two-dimensional array of binary values — *i.e.* it only contains zeros and ones, or `True` and `False`.

For instance, given the following polypetide sequences, extracted from the **CFTR** (Cystic fibrosis transmembrane conductance regulator) protein in humans and mice:

In [1]:
```
cftr_human = 'GRMMMKYRDQRAGKISERLVITSEMIENIQSVKAYCWEEAMEKMIENLRQTELKLTRKAA'
cftr_mouse = 'GKMMVKYRDQRAAKINERLVITSEIIDNIYSVKAYCWESAMEKMIENLREVELKMTRKAA'
```

we can go ahead and construct a dot matrix by comparing each residue in `cftr_human` with every residue in `cftr_mouse`.

On this occasion, D is an $n \times m$ matrix, where:

- n is equal to the number of residues in `cftr_human`; and,
- m is equal to that of `cftr_mouse`.

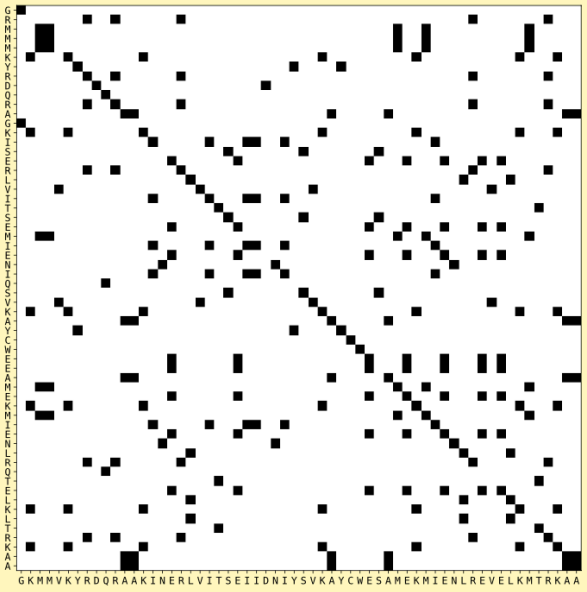
Therefore, each element in D may be referred to as $D_{i,j}$, where i and j represent the index of a residue in `cftr_human` and `cftr_mouse` respectively. The value of each element is either `0` or `1` depending on their similarity — *i.e.* if `cftr_human[i]` is identical to `cftr_mouse[j]`, then $D_{i,j}$ is `1`, otherwise it is `0`.

It is beyond the scope of this book to discuss the interpretation of dot plots. However, in general, the existence of a diagonal line in the final matrix or the dot plot would imply a high degree of similarity between the two sequence. This means that if we compare `cftr_human` with a randomly shuffled version of `cftr_mouse`, we should lose the diagonal line. To test this, we will additionally create a surrogate sequence of the `cftr_mouse` sequence:

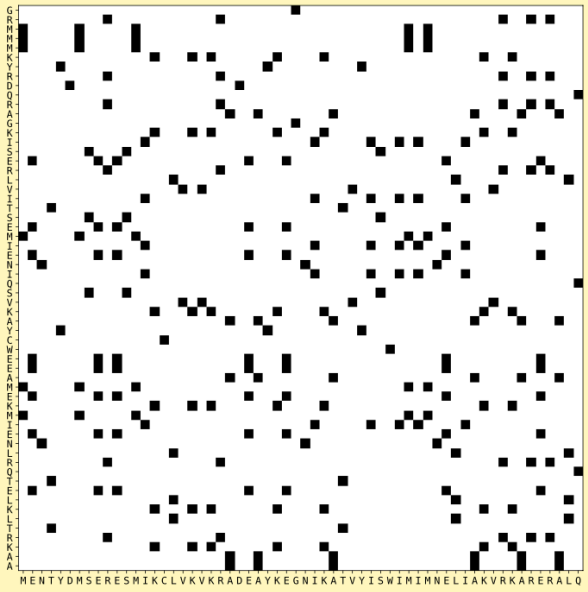
```
cftr_mouse_surrogate = 'MENTYDMSERESMIKCLVKVKRADEAYKEGNIKATVYISWIMIMNELIAKVRKARERALQ'
```

The visualisation of the dot matrices determined for the sequences we have defined are as follows, where black dots represent a value of `1` (identical residues):

`cftr_human` vs `cftg_mouse`



`cftr_human` vs `cftr_mouse_surrogate`



Implementation

We can create the above matrix by implementing a nested `for`-loop, which we then use to construct a nested array, one element at a time. This might appear somewhat difficult to grasp; however, it becomes clear once you investigate the code and the go through the workflow.

In [2]:
```
dot_matrix = list()

for residue_a in cftr_human:
    row = list()

    for residue_b in cftr_mouse:
        if residue_a == residue_b:
            row.append(1)
        else:
            row.append(0)

    dot_matrix.append(row)

print(dot_matrix)
```

To achieve a binary matrix, we compare every character in `cftr_human` to every character in `cftr_mouse`. In other words, we iterate through `cftr_human`, and during each iteration, we also iterate through `cftr_mouse`. This allows us to compare each character in `cftr_human`, to every character in `cftr_mouse`. Given that each sequence in our example has a length of 60 characters, we will essentially complete 60 iterations for `cftr_humans` (outter loop), and $60 \times 60 = 3600$ iterations for `cftr_mouse` (inner loop). This number is equal to the number of squares displayed in the above dot matrix.

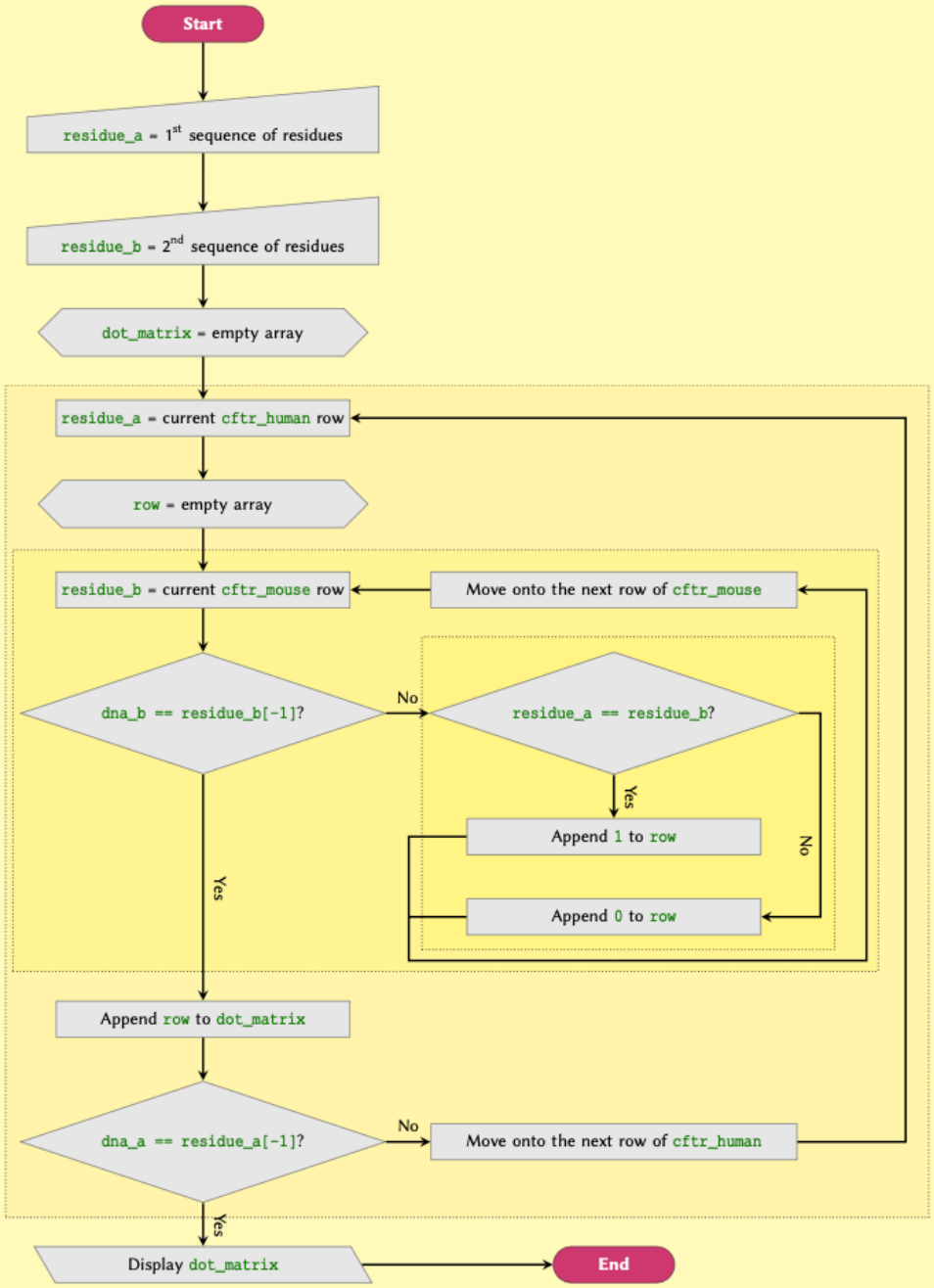
In the context of the inner loop, depending on whether or not the two characters (or amino acid residues) are identical, we append a value of `#0` or `#1` to the row. Each row represents a `list` and corresponds to the `cftr_human` residue in the current iteration. This means that each column (each item added to the row) represents one residue in `cftr_mouse`.

The output of the code will thus be a matrix of zeros and ones, as follows:

Out [2]:
```
[[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
  0, 0, 0, 0, 0, 1, 0, 0, 0], ...
```

Workflow

The algorithm that we implemented to create the dot matrix may appear relatively complex. However, it demonstrates the important and complementary relationship between `for`-loops and sequences in programming. The logic of the code is best described as a flowchart.



Visualisation

Dot plots are typically visualised as a heatmap. We do not investigate visualisation techniques here. Nonetheless, we shall take this opportunity to visualise the dot matrix we have created.

Visualisation and plotting in Python is done using external libraries, of which there are many varieties to choose from. For the purpose of this example, we will be using `LIB matplotlib`, an extensive and powerful library for producing publication quality figures in Python. The library is installed automatically as a part of the Anaconda distribution. See the official documentations, examples, and tutorials for `LIB matplotlib` to find out more.

Use the following code to produce a heatmap of the dot matrix:

In [3]:

```
from matplotlib.pyplot import subplots, show


# Initialising a figure and its axis:
fig, ax = subplots(figsize=(10, 10))

# Plotting a heatmap of the matrix with
# equal axes,  and a reversed grayscale
# colour map:
ax.imshow(dot_matrix, aspect='equal', cmap='gray_r')

# Determining the length of each sequence:
human_len, mouse_len = len(human_cftr), len(mouse_cftr)

# Creating a sequence of numbers to use
# for tick positions in our plot:
human_ticks, mouse_ticks = range(human_len), range(mouse_len)

# Setting tick positions:
ax.set_yticks(human_ticks)
ax.set_xticks(mouse_ticks)

# Setting tick labels:
ax.set_yticklabels(human_cftr, fontsize=8)
ax.set_xticklabels(mouse_cftr, fontsize=8)

# Display the figure:
show()
```

Try this code with other polypeptide or polynucleotide sequences and experiment with the `LIB matplotlib` options as described in its documentations.

LEARN MORE

To learn more about sequence comparison methods, including how to interpret dot plots, read:

- Orengo, C.A., Jones, D.T., Thornton, J.M. (2005). *Bioinformatics.* 1st ed. New York: Taylor & Francis, pp. 33-34.
- Lesk, A. (2014). *Introduction to bioinformatics.* Oxford: Oxford University Press. pp. 176-181.

while-loops

In the previous, we explored `for`-loop mediated iterations and learned that they are *exclusively* applied to iterable objects — e.g. arrays and sequences. This is because, as demonstrated in workflow figure, at the end of each iteration, the *implicit* termination condition that is inherent in the process tests whether or not it has reached the end of the sequence it is iterating through.

It may, however, be deemed necessary to apply iterative processes based on conditions other than that embedded in the `for`-loop. In such cases, we use a different class of iterations known as the `while`-loop.

REMEMBER

The termination condition of a `while`-loop is manually defined by the programmer. It is therefore paramount to ensure that the termination condition is somehow satisfied at some point in process. If the termination condition is implemented incorrectly such that the condition is never met, the iteration process would *never* end. In that case, the programme falls into what is known as the *infinite loop* state.

This is bound to happen to every programmer at some point. It is, however, important to know how to break out of an *infinite loop*:

IDE In most IDEs — *e.g.* PyCharm® or Jupyter, the programme may be terminated using the tools provided in the environment. Such tools often appear in the shape of a *stop* button in the shape of ■ or ◉, and are positioned near the *start* / *execute* button.

Terminal To abort the current task and stop the execution of a programme, we send a `SIGINT` signal in the terminal as follows:

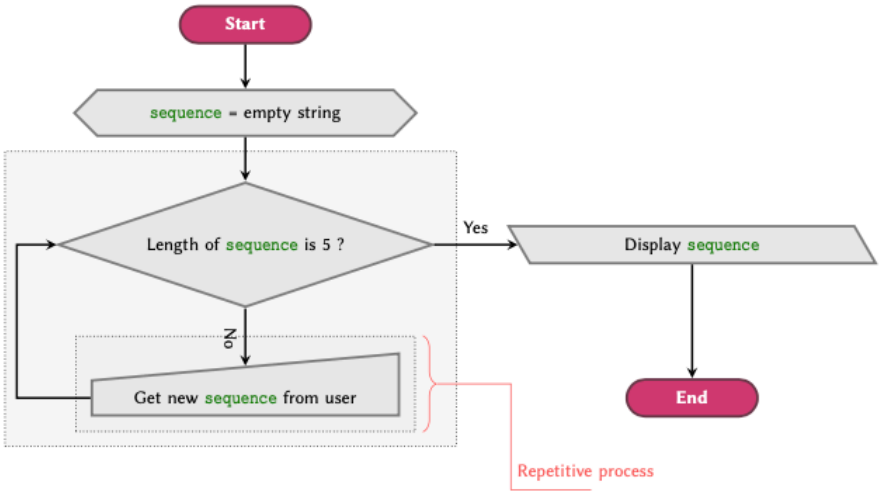
- in Linux or Microsoft Windows®, press ctrl + c
- in Mac OS X®, press ⌘ + c

Other Try to close the window manually. This will force the termination of your programme. However, if your computer freezes, the only remaining option would be to restart the computer.

Consider the following scenario:

> We want to ask the user to enter a sequence of exactly 5 amino acids in single letter code. If the provided sequence is more or less than 5 letters long, we would like to display a message and ask them to try again; otherwise, we will display the process and terminate the programme.

It is impossible to write such a process using a `for`-loop. This is because when we initialise the iteration process, the number of loops we need is unknown. In other words, we simply do not know how many times the user would need enter said sequence before they get it right.



To simplify the understanding of the concept, we can visualise the process in flowchart, as displayed in figure. In the flowchart, you can see that the only way to exit the loop is to enter a sequence of exactly 5 characters. Doing anything else — i.e. entering a different number of letters – is tantamount to going back to be beginning of the loop. The process may be described verbally as follows:

1. Initialise the variable sequence and assign an empty string to it.
2. *While* the length of sequence is not equal to 5:
    - Ask the use to enter a new sequence.
    - Go back to #2.

3. Display the value of sequence.

Implementation

We start `while`-loop using the `while` syntax, immediately followed by the loop condition.

We can now implement the process displayed in figure as follows:

```
sequence = str()

while len(sequence) != 5:
    sequence = input('Enter a sequence of exactly 5 amino acids: ')

print(sequence)
```

When executed, the above code will prompt the user to enter a value:

```
Enter a sequence of exactly 5 amino acids: GCGLLY
Enter a sequence of exactly 5 amino acids: GCGL
Enter a sequence of exactly 5 amino acids: GC
Enter a sequence of exactly 5 amino acids: GCGLL

GCGLL
```

As expected, the user is repetitively asked to enter a 5 character sequence until they supply the correct number of letters.

DO IT YOURSELF

1. Write a script which asks the user to enter a number, then:
    - if the second power of the number is smaller than 10, repeat the process and ask again;
    - if the second power of the number is equal or greater than 10, display the original value and terminate the programme.

Hint: Don't forget to convert the value enter by the user to an appropriate numeric type *before* you perform any mathematical operations.

2. We learned in subsection Sequence of Numbers that the built-in function `range()` may be utilised to produce a sequence of *integer* numbers. The function takes 3 input arguments in the following order: stop, start, step.

We now need a sequence of floating point numbers with the following criteria:

```
stop = 10
start = 0
step = 0.5
```

The use of a floating point number as step means that we cannot use `range()` to create the desired sequence. Write a script in which you use a `while`-loop to produce a sequence of floating point numbers with the above criteria and display the result.

The resulting sequence must be:

- presented as an instance of type `list`;
- similar to `range()`, the sequence must be non-inclusive — *i.e.* it must include the value of start, but not that of stop.

Solution

```
value = 0

while value ** 2 < 10:
    response = input('Enter a number: ')
    value = float(response)

print(value)
```

Solution

PYTHON

```
stop = 10
start = 0
step = 0.5

number = start
sequence = list()

while number < stop:
    sequence.append(number)
    number += step

print(sequence)
```

OUTPUT

```
[0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5]
```

Solution

PYTHON

```
# A smarter solution, however, would be:
stop = 10
start = 0
step = 0.5

sequence = [start]

while sequence[-1] < stop - step:
    sequence.append(sequence[-1] + step)

print(sequence)
```

OUTPUT

```
[0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5]
```

Breaking a while-loop

Unlike `for`-loops, it is common to *break* out of a `while`-loop mid-process. This is also known as *premature termination*.

To consider a situation that may necessitate such an approach, we shall modify our scenario as follows:

> We want to ask the user to enter a sequence of exactly 5 amino acids. If the sequence the user provides is more or less than 5 letters long, we would like to display a message and ask them to try again; otherwise, we will display the sequence and terminate the programme. Additionally, the loop should be terminated: - upon 3 failed attempts; or, - if the user entered the word exit instead of a 5 character sequence.

In the former case, however, we would also like to display a message and inform the user that we are terminating the programme because of 3 failed attempts.

To implement the first addition to our code, we will have to make the following alterations in our code:

- Define a variable to hold the iteration cycle, then test its value at the beginning of each cycle to make sure that it is below the designated threshold. Otherwise, we manually terminate the loop using the `break` syntax.
- Create a conjunctive conditional statement for the `while`-loop to make so that it is also sensitive to our exit keyword.

```
sequence = str()
counter = 1
max_counter = 3
exit_keyword = 'exit'

while len(sequence) != 5 and sequence != exit_keyword:
    if counter == max_counter:
        sequence = "Three failed attempt - I'm done."
        break

    sequence = input('Enter a sequence of exactly 5 amino acids or [exit]: ')

    counter += 1

print(sequence)
```

Exercises

END OF CHAPTER EXERCISES

1. Can you explain the reason why in the example given in subsection for-loop and conditional statements we set minimum to be equal to the first value of our array instead of, for instance zero or some other number?

Store your answer in a variable and display it using `print()`.

2. Write a script that using a `for` loop, calculates the sum of all numbers in an array defined as follows:

```
numbers = [0, -2.1, 1.5, 3]
```

and display the result as:

```
Sum of the numbers in the array is 2.4
```

3. Given an array of integer values as:

```
numbers = [2, 1, 3]
```

write a script that using `for` loops, displays each number in the list as many time as the number itself. The programme must therefore display 2 twice, 1 once, and 3 three times.

4. Given a `list` of numbers defined as:

```
numbers = [7, 16, 0.3, 0, 15, -4, 5, 3, 15]
```

write a script that using at most two `for` loops, finds the *variance* of the numbers, and display the **mean**, and the **variance**. Note that you will need to calculate the *mean* as a part of your calculations to find the *variance*.

The equation for calculating the variance is:

$$\sigma^2 = \frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}$$

Hint: Breaking down the problem into smaller pieces will simplify the process of translating it into code and thereby solving it:

a. Work out the Mean or μ (the simple average of the numbers):

$$\mu = \frac{\sum_{i=1}^{n} x_i}{n}$$

b. Calculate the sum of: each number (x_i) subtracted by the Mean (μ) and square the result.

c. Divide the resulting number by the length of number.

Display the results in the following format:

```
Mean: XXXX
Variance: XXXX
```

Solution

Q1

PYTHON

```
answer = "Because the minimum of the array may be smaller than zero."

print(answer)
```

OUTPUT

```
Because the minimum of the array may be smaller than zero.
```

Q2

PYTHON

```
numbers = [0, -2.1, 1.5, 3]

numbers_sum = 0

for value in numbers:
    numbers_sum += value

print("Sum of the numbers in the array is", numbers_sum)
```

OUTPUT

```
Sum of the numbers in the array is 2.4
```

Q3

PYTHON

```
numbers = [2, 1, 3]

for value in numbers:
    prepped_value = [value] * value

    for number in prepped_value:
        print(number)
```

OUTPUT

```
2
2
1
3
3
3
```

Q4

PYTHON

```python
numbers = [7, 16, 0.3, 0, 15, -4, 5, 3, 15]

numbers_length = len(numbers)

# Calculating the "sum"
# ------------------------------------------------------------
numbers_sum = 0

for value in numbers:
    numbers_sum += value


# Calculating the "mean"
# ------------------------------------------------------------

numbers_mean = numbers_sum / numbers_length


# Calculating the "variance"
# ------------------------------------------------------------

variance_denominator = numbers_length
variance_numerator = 0

for value in numbers:
    prepped_value = (value - numbers_mean) ** 2
    variance_numerator += prepped_value

numbers_variance = variance_numerator / variance_denominator


# Results
# ------------------------------------------------------------

print("Mean:", numbers_mean)
print("Variance:", numbers_variance)
```

OUTPUT

```
Mean: 6.366666666666666
Variance: 48.919999999999995
```

KEY POINTS

- Iterations and loops are used to perform repetitive operations.
- Implementation of `for`-loop involves 4 steps.
- Conditional statements are used within loops to handle different situations.
- `while`-loop is suited when exact number of conditions/iterations are unknown.