Chapter 1: Storing Sequences In Python

PART 1: Storing a single DNA base

A single DNA base can be stored in Python as follows:

a = 'T'

Here I have created a variable named as **a** and I have assigned a value 'T' to it.

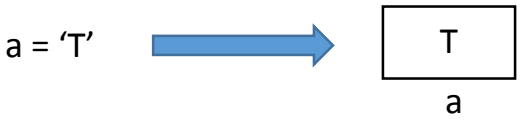
So let's see what a variable is:

i) A variable is something whose value can change throughout the program.

ii) A variable is a name given to a memory location that stores some data.

We can think of it as follows:

When I write a = 'T', a box gets created in the computer memory, and then we store the character 'T' in it, and then we name this box as a.



This satisfies our second statement about variable that says that it is a name given to a memory location and stores some data.

Now if I write a = 'G', this will cause the value inside the box to be replaced from 'T' to 'G'. So now variable **a** will store the character 'G' instead of 'T'. And this satisfies our first statement about variable which says that value of a variable can change throughout the program.

We create a variable in Python by writing, variable name followed by an equal to sign which is followed by the value that we want to store in that variable.

Note: variable name could be anything meaningful, the only restrictions are that a variable should not start with a number and a variable name cannot consist of any special characters except underscore i.e. '_'.

So in the current example, when I write a = 'T', **a** is the variable name, then we have '=' symbol and then character 'T' is the value that we want to store in this variable.

Note: = is also known as the assignment operator as it is used to assign value to a variable.

Now we know how to create a variable and store a single DNA base in it.

If we want to print the value of a variable, then in Python we make use of the print() function and write it as follows:

a = 'C'

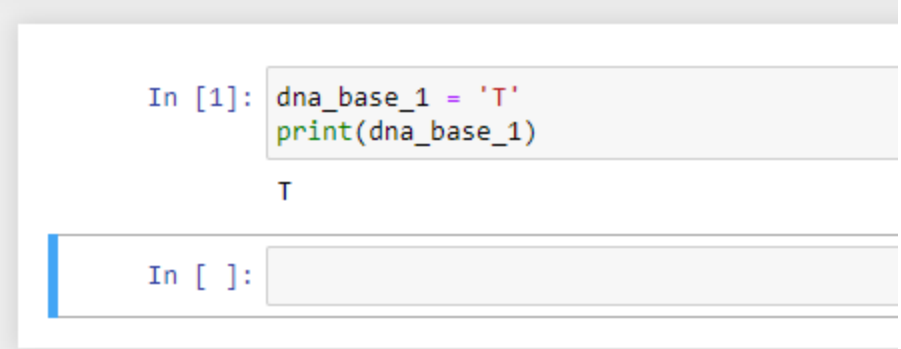
print(a)

The above code will print **C** in the console.

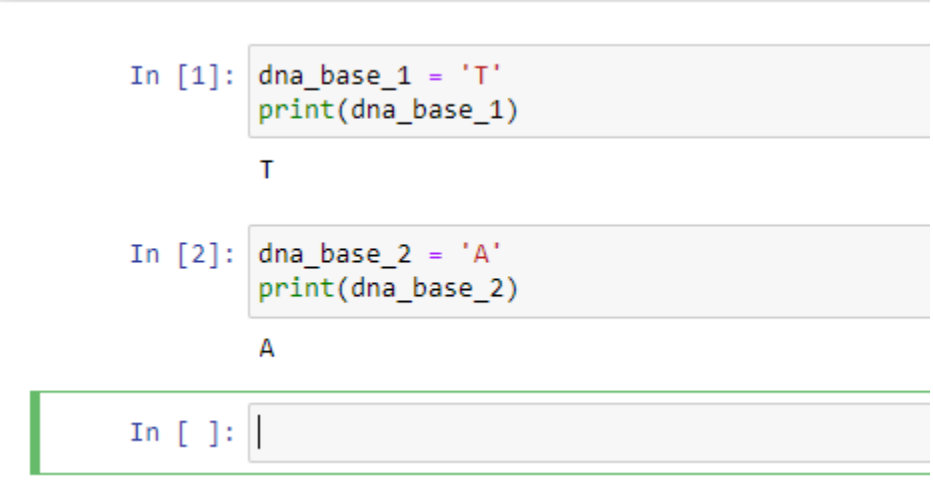
Now let's take a look at the following tasks and complete them in Jupyter Notebook.

Note: Jupyter Notebook is an IDE (Integrated Developmet Environment), which is a text editor that allows us to write and execute Python code.

1.) Create a variable named as dna_base_1 and store base T in it. Also print the value of variable dna_base_1.



2.) Create a variable named as dna_base_2 and store base A in it. Also print the value of the variable dna_base_2.



3.) Create a variable named as dna_base_3 and store base G in it. Also print the value of the variable dna_base_3.

```
In [2]: dna_base_2 = 'A'
        print(dna_base_2)

        A

In [3]: dna_base_3 = 'G'
        print(dna_base_3)

        G

In [ ]:
```

4.) Create a variable named as dna_base_4 and store base C in it. Also print the value of the variable dna_base_4.

```
In [3]: dna_base_3 = 'G'
        print(dna_base_3)

        G

In [4]: dna_base_4 = 'C'
        print(dna_base_4)

        C

In [ ]:
```

Here we have created 4 different variables and stored each DNA base in it.

We can also do the same in one single cell of Jupyter Notebook as follows:

```
In [4]: dna_base_4 = 'C'
        print(dna_base_4)

        C

In [5]: dna_base_1 = 'T'
        dna_base_2 = 'A'
        dna_base_3 = 'G'
        dna_base_4 = 'C'

        print(dna_base_1)
        print(dna_base_2)
        print(dna_base_3)
        print(dna_base_4)

        T
        A
        G
        C
```

Note: You must have noticed that whenever I want to create a variable name that has multiple words, I make use of underscore i.e. '_' to join the individual words. This is a convention and I would encourage you to do the same, whenever you create variables having long variable names with multiple words.

Important: You should never use spaces in between variable names, otherwise you will end up getting the following error:

```
In [8]: dna base 1 = 'T'
        print(dna base 1)

          File "C:\Users\dell\AppData\Local\Temp\ipykernel_572\4213839104.py", line 1
            dna base 1 = 'T'
                     ^
        SyntaxError: invalid syntax


In [ ]:
```

In the same way we can also store and print amino acids.

```
In [6]: alanine = 'A'
        arginine = 'R'
        asparagine = 'N'
        aspartic_acid = 'D'

        print(alanine)
        print(arginine)
        print(asparagine)
        print(aspartic_acid)
```

```
A
R
N
D
```

Here variable names are amino acid full names and the values being stored in the variables are amino acid single letter codes.

And in the same way we can store any single character in a variable in Python.

Here are some examples:

```
In [7]: asterick = '*'
        star = '*'
        space = ' '
        dollar = '$'

        print(asterick)
        print(star)
        print(space)
        print(dollar)
```

```
*
*

$
```

Pay attention to how I have stored a single space in the variable named as **space.** This is to make you realize that a space is also a single character even though it is not visible when it is printed. If you are finding this confusing don't worry this will get clearer as we proceed further.

Till now we know that a variable stores some data. But this data that gets stored in a variable can be of multiple types. Formally the type of the data stored in a variable is known as the **data type** of that variable.

Python has the following data types: Numeric, String, List, Dictionary, Tuple and Set. For now just remember their names. We will be looking into the details for each one of them as we proceed.

Now if we go back to the first example in which we created a variable named as **a** and stored base T in it.

So if we want to find out the type of **T** or we can say if we want to find out the data type of variable **a,** then we can make use of type() function.

```
In [1]: a = 'T'
        print(a)
        print(type(a))

        T
        <class 'str'>

In [ ]:
```

In the above, in the first line we have created the variable. In the next line we are using print() function to print the value of the variable and in the last line we are using the type() function inside the print() function to print the data type of this variable.

In the output it shows that the data type of variable **a** is <class 'str'>. This means that the value stored inside the variable **a** is of **String** type**.** Which further means that base or character T is a String.

Now the question arises what exactly is a String?

Before understanding Strings we must first understand what a Character is**.** A character is any key that is present on our keyboard. Alphabets from A-Z and a-z, space, special characters such as *, !, #, $, %, etc, are all considered as characters.

And a String is defined as a sequence of characters enclosed within single or double quotes.

This means that even if we have a single character such as a DNA base, and it is enclosed within single or double quotes, then it is a String. This is the reason why for all the variables that we created till now, I enclosed the DNA Base or character within quotes.

Now let's create some variables and check their type.

```
In [3]: dna_base_1 = 'T'
        dna_base_2 = 'A'
        dna_base_3 = 'G'
        dna_base_4 = 'C'

        print(dna_base_1)
        print(type(dna_base_1))

        print(dna_base_2)
        print(type(dna_base_2))

        print(dna_base_3)
        print(type(dna_base_3))

        print(dna_base_4)
        print(type(dna_base_4))
```

```
T
<class 'str'>
A
<class 'str'>
G
<class 'str'>
C
<class 'str'>
```

```
In [ ]:
```

We can see that all of the variables are of type String.

Some non-biological examples of Strings having a single character are as follows:

```
In [5]: asterick = '*'
        hashtag = '#'
        space = ' '
        dollar = '$'

        print(asterick)
        print(type(asterick))

        print(hashtag)
        print(type(hashtag))

        print(space)
        print(type(space))

        print(dollar)
        print(type(dollar))
```

```
*
<class 'str'>
#
<class 'str'>

<class 'str'>
$
<class 'str'>
```

An important thing to notice here is that a single space enclosed within quotes (third variable) is also a String.

Note: It doesn't matters if we are using single or double quotes. Try creating all the above variables using double quotes instead of single quotes and see if there is any difference.

Now as I mentioned earlier that a String is a sequence of characters enclosed within quotes, therefore we can also stores sequence of characters as a String. For example, if I want to store my name (which is nothing but a sequence of characters) in a variable in Python, I will write it as follows:

```
In [6]: my_name = 'Ashish Singh'
        print(my_name)
        print(type(my_name))

        Ashish Singh
        <class 'str'>

In [ ]:
```

Here I have created a variable named as my_name and I have stored my name i.e Ashish Singh into this variable. And we can see after printing its type that this is also a String.

Some other random examples of Strings having a sequence of characters are as follows:

```
In [7]: s1 = 'What is your name?'
        s2 = 'I have 2 apples in my basket.'
        s3 = 'If I add 3 and 12, I will get the answer as 15!'

        print(s1)
        print(type(s1))

        print(s2)
        print(type(s2))

        print(s3)
        print(type(s3))

        What is your name?
        <class 'str'>
        I have 2 apples in my basket.
        <class 'str'>
        If I add 3 and 12, I will get the answer as 15!
        <class 'str'>

In [ ]:
```

To conclude, now we know how to store a single DNA base, amino in Python and we also know that is gets stored as a String.

PART 2: Storing a single DNA sequence

With all the things that we have learnt so far we can say that a single DNA sequence will also get stored in Python as a String.

This will also be true for a Protein sequence or RNA sequence.

This is because a DNA, RNA or Protein sequences are nothing but sequence of characters.

For example a DNA sequence is simply a sequence of characters A, T, G and C.

Now let's create some variables and store DNA sequences in them:

```
In [8]: dna_1 = 'ATGC'
        print(dna_1)
        print(type(dna_1))

        dna_2 = 'ATAGCAGTACGACATCAGCATCGACTACAGCATAGCATCAGCA'
        print(dna_2)
        print(type(dna_2))
```

```
ATGC
<class 'str'>
ATAGCAGTACGACATCAGCATCGACTACAGCATAGCATCAGCA
<class 'str'>
```

```
In [ ]:
```

Let's create some more variables and store RNA and Protein sequences in them:

```
In [9]: rna_1 = 'AUGCAGACGAUAGACGAGU'
        print(rna_1)
        print(type(rna_1))

        protein_1 = 'CQDNRTDEKFS'
        print(protein_1)
        print(type(protein_1))
```

```
AUGCAGACGAUAGACGAGU
<class 'str'>
CQDNRTDEKFS
<class 'str'>
```

```
In [ ]:
```

As we can see all of them are Strings.

Now we need to learn about a very important characteristic of Strings that is Sequential Memory Allocation. What this means is that whenever we create a String, each character of that String is allocated memory in a sequence.

Let's understand this by taking a deeper look at what happens when we create a String in computer memory.

s1 = 'Python for Bioinformatics' # here I have created a String 'Python for Bioinformatics' and named it as s1.

As soon as I execute this line of code, a series of boxes get created inside computer memory and each of the character gets stored in each box, as follows:

P	y	t	h	o	n		f	o	r		B	i	o
0	1	2	3	4	5	6	7	8	9	10	11	12	13

All these boxes are created in a sequence and thus are numbered in a sequence, starting from 0.

Character 'P' has an index 0

Character 'y' has an index 1

Character 't' has an index 2

And so on, till we reach the last character 'o' having index 13.

Another important thing to notice here is that since spaces are also characters therefore they are also assigned index.

The first space after character 'n' has an index of 6, and the second space after character 'r' has an index 10.

This indexing also happens in the opposite direction, starting from -1, as follows:

P	y	t	h	o	n		f	o	r		B	i	o
-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

In the reverse direction:

Character 'o' has an index of -1

Character 'i' has an index of -2

Character 'B' has an index of -3

And so on, till the first character 'P' this is having index of -14

So this indexing on each character of a String, in a sequential manner, starting from 0 in the forward direction and starting from -1 in the reverse direction, is knows as Sequential Memory Allocation of Strings.

We make use of this indexing, when we try to fetch individual character or a range of characters from our string.

Now let's take an example of a random DNA sequence and understand how it is indexed.

```
In [1]: # here I am creating a random DNA sequence
        dna_1 = "ATGC"

        # printing the variable
        print(dna_1)

        # printing the type of the variable
        print(type(dna_1))

        ATGC
        <class 'str'>
```

Nothing much happening in the above code, just printing the value and type of the variable.

Now let's say I want to fetch only the first base from out dna_1 string. This is how the code will look:

```
In [2]: first_base = dna_1[0]
        print(first_base)

        A

In [ ]:
```

So, in order to fetch the first base, in the above code, we write 0 inside square brackets, because the index of the first base is 0.

Syntax: variable_name[index]

The general syntax for fetching a single character from a string is writing the variable name followed by square brackets and then mention the index of that character within the square brackets.

In the above example, I have written dna_1[0] since, the name of the variable is dna_1 and the character that I want to fetch is present at index 0.

I went further and stored the first base into another variable named as first_base and then printed the value for first_base.

Storing the fetched value into another variable totally depends on your requirement. If you don't want to store the fetched value and just want to print it, you can simply write it inside a print function. Take a look at the third line in the following code:

```
In [3]: first_base = dna_1[0]
        print(first_base)
        print(dna_1[0])

        A
        A

In [ ]:
```

Now we will do some practice on fetching different individual bases from a DNA sequence. Remember that this will also work for Protein and RNA sequences.

```
In [4]: dna_1 = "ATGAATAGACGATAGCATGCAGT"

        # fetching the first base
        first_base = dna_1[0]

        # fetching the third base
        third_base = dna_1[2]

        # fetching the fifth base
        fifth_base = dna_1[4]

        print(first_base)
        print(third_base)
        print(fifth_base)

        A
        G
        A

In [ ]:
```

And as we learnt that this indexing of characters also occurs in the reverse direction starting from -1, so we can also make use of negative indexing for fetching the DNA bases.

For example we can fetch the last base from our DNA sequence in the following two ways:

```
In [6]: # fetching the last base using positive index
        last_base = dna_1[22]
        print(last_base)

        # fetching the last base using negative index
        last_base = dna_1[-1]
        print(last_base)
```

```
T
T
```

```
In [ ]:
```

In the above code you can see how we made use of negative index to fetch the last character. Using the negative index for fetching the last DNA base is more useful because in practice our DNA sequences will be very long and it will be difficult to determine the positive index of the last base, whereas using a negative index makes it very easy to fetch the last base, because no matter how long the DNA sequence is, the last base will always be present at index -1.

Now we will take a look at how to fetch a sub sequence from our DNA sequence. If I want to fetch first three bases from my DNA sequence, it is done as follows:

```
In [11]: dna_1 = "ATGAATAGACGATAGCATGCAGT"
         print(dna_1)
         first_three_bases = dna_1[0:3]
         print(first_three_bases)
```

```
ATGAATAGACGATAGCATGCAGT
ATG
```

```
In [ ]:
```

Syntax: variable_name[starting_index : ending_index]

The syntax is similar to fetching single bases, but now instead of a single index, we give a range within the square brackets, the first parameter is the starting index, then we put a colon and then we give the second parameter that is the ending index.

So in the above example, for fetching the first three bases, I passed the starting index as 0 and the ending index as 3. Now you might question why I used 3 instead of 2 as the ending index, because the third base G is actually present at index 2.

The reason for this is that in Python language the ending index is not counted or you can say that the ending index gets excluded. So when I write dna_1[0:3], Python starts counting from 0 but it only counts till 2 and therefore we get the bases that are present at the index 0, 1 and 2.

If I would have written dna_1[0:2], then it will only give me that bases at the index 0 and 1, i.e. the first two bases. So basically the ending index that we pass within square brackets has to be 1 more than the actual index of the character till where we want to fetch. This is get clearer in the coming examples.

This process of fetching subsequence from a DNA sequence (String) is knows as **Slicing**, as we are basically taking a slice from our bigger String.

Now let's look at few more examples of slicing our String or DNA sequence:

```
In [14]: dna_1 = "ATGAATAGACGATAGCATGCAGT"
         print(dna_1)

         # fethcing base from index 2 to index 5
         print(dna_1[2:6])

         # fetching base from 2nd position to 5th position
         print(dna_1[1:5])

         ATGAATAGACGATAGCATGCAGT
         GAAT
         TGAA
```

In the above example first we are fetching bases from index 2 to index 5. If we look at our DNA sequence, then bases from index 2 to index 5 are GAAT, because base G is at index 2 (∵ indexing starts from 0) and base T is at index 5. Now to fetch them, I passed 2 as the starting index and 6 as the ending index, because then only Python will count from index 2 to index 5. (∵ ending index gets excluded).

In the next line we are fetching bases from 2nd position to 5th position. Now this is not same as the first example. Here if we look at the DNA sequence, bases from 2nd position to 5th position are TGAA, because T is at second position and A is at 5th position. Now to fetch them, I passed:

a.) starting index as 1, since second position corresponds to index 1 (∵ indexing starts from 0)

b.) ending index as 5. Now, the 5th position corresponds to index 4, but to make Python count till index 4, we need to pass ending index as 5 (∵ ending index gets excluded).

Let's take another example on String/DNA slicing:

```
In [22]: dna_1 = "ATGAATAGACGATAGCATGCAGT"
print(dna_1)

# not providing the starting and the ending index
print(dna_1[:])

# providing only the starting index
print(dna_1[7:])

# providing only the ending index
print(dna_1[:10])
```

```
ATGAATAGACGATAGCATGCAGT
ATGAATAGACGATAGCATGCAGT
GACGATAGCATGCAGT
ATGAATAGAC
```

In the above example, the first case is where we neither provide the starting index, nor the ending index and then in the output we get the complete DNA sequence.

This means that in Python, if we don't pass any index while slicing then by default the starting index is taken as 0, and ending index is taken as the index of the last character + 1 i.e. till the very end of our sequence.

In the next case, we only provide the starting index as 7, therefore we get the bases starting from the 7^{th} index right till the very end of our DNA sequence.

In the last case we only provide the ending index as 10, therefore we get the bases starting from index 0 till 9^{th} index, or we can say, we are getting the first 10 bases.

This defaulting of the indexes become very useful if we want to fetch bases from the last part of our DNA sequence.

Let's take a look at the below example:

```
In [15]: dna_1 = "ATGAATAGACGATAGCATGCAGT"
print(dna_1)

# fethcing the last 4 bases
print(dna_1[19:23])
```

```
ATGAATAGACGATAGCATGCAGT
CAGT
```

```
In [ ]:
```

In the above example we are trying to fetch the last 4 bases from our DNA sequence using positive indexes. Again if the DNA sequence is too long, it will be very difficult for us to find out the index of the 4th last base. Therefore, in this case we can make use of negative indexes as follows:

```
In [27]: dna_1 = "ATGAATAGACGATAGCATGCAGT"
         print(dna_1)

         # fethcing the last 4 bases using positive indexes
         print(dna_1[19:23])

         # fetching the last 4 bases using negative indexes
         print(dna_1[-4:])
```

```
ATGAATAGACGATAGCATGCAGT
CAGT
CAGT
```

Here we give the starting index as -4 for the fourth last base, and we don't give any ending index, because we know that by default it will return us the sequence till the very end. This method is useful because no matter how long our DNA sequence is, the negative index for fourth last base will always be -4 and simply leaving the ending index blank will give us the last 4 bases of our DNA sequence.

Similarly we can fetch last ten bases as follows:

```
In [28]: dna_1 = "ATGAATAGACGATAGCATGCAGT"
         print(dna_1)

         # fethcing the last 10 bases using negative indexes
         print(dna_1[-10:])
```

```
ATGAATAGACGATAGCATGCAGT
AGCATGCAGT
```

So till now while slicing Strings/DNA we were only passing starting and the ending parameter. But there is also one more additional that we can pass, and I like to call it the skipping parameter. Let's understand this with the help of examples.

```
In [31]: dna_1 = "ATGAATAGACGATAGCATGCAGT"
         print(dna_1)

         # fetching the complete DNA sequence with skipping parameter set to 1
         print(dna_1[0:23:1])

         # fetching the complete DNA sequence with skipping parameter set to 2
         print(dna_1[0:23:2])

         # fetching the complete DNA sequence with skipping parameter set to 3
         print(dna_1[0:23:3])
```

```
ATGAATAGACGATAGCATGCAGT
ATGAATAGACGATAGCATGCAGT
AGAAAGTGAGAT
AAACTCGG
```

```
In [ ]:
```

In the above example in the first case we are simply fetching the entire DNA sequence, but we have also passed an additional third parameter i.e. the skipping parameter as 1.

Passing the skipping parameter as 1 shows no effect on the output. This means that the default value for skipping parameter is 1.

This means that whenever we are just passing the starting and ending parameters and not passing any value for the skipping parameter while slicing then by the default the value for the skipping parameter is taken as 1.

In the second case, we have set the skipping parameter to 2, and now you can see that while slicing we are skipping one base and printing the next one. So base A gets printed, T gets skipped, G gets printed, A gets skipped, the next A gets printed, T gets skipped and so on.

In the thirds case, skipping parameter has been set to 3, and therefore it prints a base, then skips the next two bases and then prints the next one. So, base A gets printed, T and G gets skipped, next A gets printed, A and T gets skipped, next A gets printed and so on.

The above three cases can also be written without explicitly passing the starting and the ending indexes as follows:

```
In [32]: dna_1 = "ATGAATAGACGATAGCATGCAGT"
         print(dna_1)

         # fetching the complete DNA sequence with skipping parameter set to 1
         print(dna_1[::1])

         # fetching the complete DNA sequence with skipping parameter set to 2
         print(dna_1[::2])

         # fetching the complete DNA sequence with skipping parameter set to 3
         print(dna_1[::3])
```

```
ATGAATAGACGATAGCATGCAGT
ATGAATAGACGATAGCATGCAGT
AGAAAGTGAGAT
AAACTCGG
```

```
In [ ]:
```

Here we are not passing any starting or ending index, because we know that by default they will be taken as the start and end of our DNA sequence.

And if we don't pass any parameter while slicing, then it will simply return us the complete sequence with no change. This can be seeing in the following code:

```
In [34]: dna_1 = "ATGAATAGACGATAGCATGCAGT"
         print(dna_1)

         # fetching the complete DNA sequence
         print(dna_1[::])
```

```
ATGAATAGACGATAGCATGCAGT
ATGAATAGACGATAGCATGCAGT
```

```
In [ ]:
```

Now just like starting and ending indexes, skipping parameter also has both positive and negative values.

```
In [53]: dna_1 = "ATGAATAGACGATAGCATGCAGT"
         print(dna_1)

         # fetching the first 10 bases with skipping parameter set to -1
         print(dna_1[10::-1])

         ATGAATAGACGATAGCATGCAGT
         GCAGATAAGTA

In [ ]:
```

In this example we have set the starting index as 10, left the ending index blank and skipping parameter is -1.

Now whenever the skipping parameter has a negative value, our String or DNA sequence will be read in the reverse direction.

Since, the starting index is 10, we start at the 10^{th} index, and since there is no ending index, therefore it will read till the very end of the DNA sequence, but this will happen in the opposite direction, because skipping parameter is -1.

This means that we will start at the 10^{th} index we will start moving towards left till we reach the very first base i.e. 0^{th} index and therefore we get first 10 bases in reversed order in the output.

Now this feature is very helpful whenever we want to reverse our DNA sequences.

```
In [63]: dna_1 = "ATGAATAGACGATAGCATGCAGT"
         print(dna_1)

         # reversing the DNA sequence
         print(dna_1[::-1])

         ATGAATAGACGATAGCATGCAGT
         TGACGTACGATAGCAGATAAGTA

In [ ]:
```

In the above example we are reversing our DNA sequence, notice that I have just passed the skipping parameter as -1 and have not passed any starting and ending index. In this situation, when skipping parameter is -1 and we don't have any values for starting and ending index, then by default the starting index points to the end of the String or DNA sequence and ending index points to the start of the string or DNA sequence.

The best thing here is that we can simply reverse in Python to reverse any DNA, Protein or RNA sequence, just by using slicing and that too in a single line.