Working with Numbers and Strings

In programming a distinction is made between **numbers** and **text**.

A number can be printed as is.

A string is a bunch of text characters

and to distinguish **text** from **numbers**

we surround it with a **quotes**

Let's print some numbers and strings

In [5]:
```
1 + 5
```

Out[5]: 6

In [6]:
```
# this will either give an error or only read the last line (depending on your IDE)

1 + 5
+ 3
```

Out[6]: 3

In [7]:
```
# you may use brackets to show that you are not done with your statement

(1 + 5
+ 3)
```

Out[7]: 9

In [8]:
```
# you may also use an escape character "\"
# to show that you are not done with your statement

1 + 5 \
 + 3
```

Out[8]: 9

Warning!!

Be careful what follows the `"\"`

Because there are also special characters, such as:

`\n` = new line

`\t` = tab

....etc.

This may make your code do someting you did not intend

Some Python operations include:

Symbol	Task Performed
+	Addition
-	Subtraction
/	division
%	modulo
*	multiplication
//	floor division
**	to the power of

> Note that some of these can be used with strings and numbers

> Other mathematical and statistical operations may be imported using Python libraries

In [9]:
```
# this output a float, because it gives a decimal number

10/4
```

Out[9]: 2.5

In [10]:
```
# Floor (//)
# Formal definition: the largest integer not greater than the result of the division
# Similar to rounding DOWN

10 // 4
```

Out[10]: 2

In [11]:
```
round (10/4)
```

Out[11]: 2

In [12]:
```
# we can also import a Math library
# which gives us additional capabilities

import math

math.floor(10/4) # rounds down
```

Out[12]: 2

In [13]:
```
math.ceil(10/4) #rounds up
```

Out[13]: 3

In [14]:
```
# returns the remainder after division

10%4 # 10 /4  = 8 , with a remainder of 2
```

Out[14]: 2

In [15]:
```
10%3 # 10 /3  = 9 , with a remainder of 1
```

Out[15]: 1

Note that any decimal number is called a float

A non-decimal number is an integer

To convert between number types

You may use the built-in functions

`int ()` and `float()`

In [16]:
```
# e.g.

int (2.5)
```

Out[16]: 2

In [17]:
```
float (2)
```

Out[17]: 2.0

But you can also turn numbers into text

Although its use might not be immediately apparent

In [18]:
```
# note that no math is being done. This is now a string

'1 + 5'
```

Out[18]: '1 + 5'

In [19]:
```
#these two are not the same
str(1 + 5)
```

Out[19]: '6'

> Although the above statement doesn't give an error

> This only really works in this way in Jupyter

> It's better to do it properly

> Using the print function

In [3]:
```
# print () is a function, which accepts an argument inside the brackets
print()
```

In [21]:
```
print ('1 + 5')
```

```
1 + 5
```

> Note that you have to use **BRACKETS** and you have to use **QUOTES**

In [22]:
```
print (I am learning to code)
```

```
  File "<ipython-input-22-d9d24abb22ce>", line 1
    print (I am learning to code)
               ^
SyntaxError: invalid syntax
```

In [1]:
```
print ("I am learning to code") # this entire line is called a STATEMENT
```

```
I am learning to code
```

In [2]:
```
print ('I am learning to code') # to print strings, you may use single or double quotes
```

```
I am learning to code
```

Normally we assign strings and numbers to **variables**

A variable is like a file or folder that hold your important information

Normally, you would name your file or folder in such a way

That is is easy for you to know what is kept inside it

By looking at the file or folder name

In [ ]:
```
first_number = 1
second_number = 2
add_two_numbers = first_number + second_number
# note that this last statement will NOT print
# unless you explicitly use the print function
```

In [ ]:
```
print (add_two_numbers)
```

Rules for Choosing Variable Names

- No spaces or tabs at the start of the variable name OR in between the variable name, e.g. `first number`

- Indentation matters in Python, so start at the start of the line

- You may use built-in Python words or booleans # e.g. `print` or `True`

- When you use a variable name twice, it takes the new value

- If you use `#` at the start of the line, the line will not be executed. It's a signal to say: "I just want to leave a comment."

- Use a meaningful names, so that your code is easy to read

In [4]:
```
my_dna = "ATGCGTA" # you have now assigned a sequence of DNA to your variable name `my_dna`
print (my_dna)
```

```
ATGCGTA
```

In [5]:
```
my_dna = "TGGTCCA" # if you use the same variable name, your first sequence is now lost
print (my_dna)
```

```
TGGTCCA
```

Concatenation and Multiplication

In [6]:
```
dna_1 = "ATGCGTA"
dna_2 = "TGGTCCA"
add_dna1_dna2 = dna_1 + dna_2
print (add_dna1_dna2)
```

```
ATGCGTATGGTCCA
```

In [7]:
```
poly_A_tail = "AAA"*3

print (poly_A_tail)
```

```
AAAAAAAAA
```

The `len ()` Function

In [8]:
```
# What is the length of your DNA sequence
len(dna_1)
```

Out[8]:
```
7
```

In [ ]:
```
len(add_dna1_dna2)
```

In [9]:
```
# this will give an error


print ("The length of my 1st dna sequence", dna_1, "is:" +  len(dna_1))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-2de9cde2c7d1> in <module>
      2
      3
----> 4 print ("The length of my 1st dna sequence", dna_1, "is:" +  len(dna_1))

TypeError: can only concatenate str (not "int") to str
```

In [10]:
```
# using a comma

print ("The length of my dna sequence", dna_1, "is:",  len(dna_1))
```

```
The length of my dna sequence ATGCGTA is: 7
```

In [11]:
```
# str() is another built in function to convert a number to a string

print ("The length of my dna sequence", dna_1, "is:" +  str(len(dna_1)))
```

```
The length of my dna sequence ATGCGTA is:7
```

In [12]:
```

string_of_length_dna_1 = str(len(dna_1))
print ("The length of my dna sequence", dna_1, "is:" +  string_of_length_dna_1)
# note that when using commas, a space is automatically inserted, but not when using a "+"
```

```
The length of my dna sequence ATGCGTA is:7
```

In [13]:
```
# %s

print ("The length of my dna sequence", dna_1, "is: %s"  %len(dna_1))


```

```
The length of my dna sequence ATGCGTA is: 7
```

There are many other such operators

But for now, just keep them in mind

- %s -> string
- %d -> Integer
- %f -> Float
- %o -> Octal
- %x -> Hexadecimal
- %e -> exponential

Changing case

To change between cases, we can two methods

A method is like a function, but it normally belongs to a certain data type in programming languages, and does not work with other data types

For example, you may use the `print ()` function to print both numbers and strings, but you may not use the methods `.upper()` and `.lower()` with numbers.

> Normally methods also use a `.` before the method name

> And with methods, you usualy do not add an **argument** inside the brackets. That is, `.lower()` remains empty

In [14]:
```
my_dna = "ATGC"
```

In [15]:
```
# print my_dna in lower case
print(my_dna.lower())
```

```
atgc
```

> Also note, that with **Methods**, the **variable name** comes **first** and then the method is applied: `my_dna.lower()`

> While with **Functions**, the **function name** comes **first**, and then the variable name: `print(my_dna)`

Replacement

> The `.replace()` method takes **two arguments**

In [16]:
```
old_boyfriend = "Jack"
print (old_boyfriend)
```

```
Jack
```

In [17]:
```
# replace "ck" with "ke"
new_boyfriend = old_boyfriend.replace("ck", "ke")
print (new_boyfriend)
```

```
Jake
```

In [18]:
```
# you may replace multiple characters
# or even replace it with nothing, a.k.a. delete characters

protein = "vlspadktnv"

print(protein.replace("pad",""))
print(protein.replace("pad"," ")) # note that there is a difference between this and the above
```

```
vlsktnv
vls ktnv
```

Extracting part of a string

> We also use square brackets [ ] to deal with sub-sections of the string ***(substring)***

In [19]:
```
protein = "vlspadktnv"
# print positions three to five
print(protein[3:5])
```

```
pa
```

Warning!!

Note that the first position is counted as position zero and not as position 1

Aso note that the positions are inclusive at the start

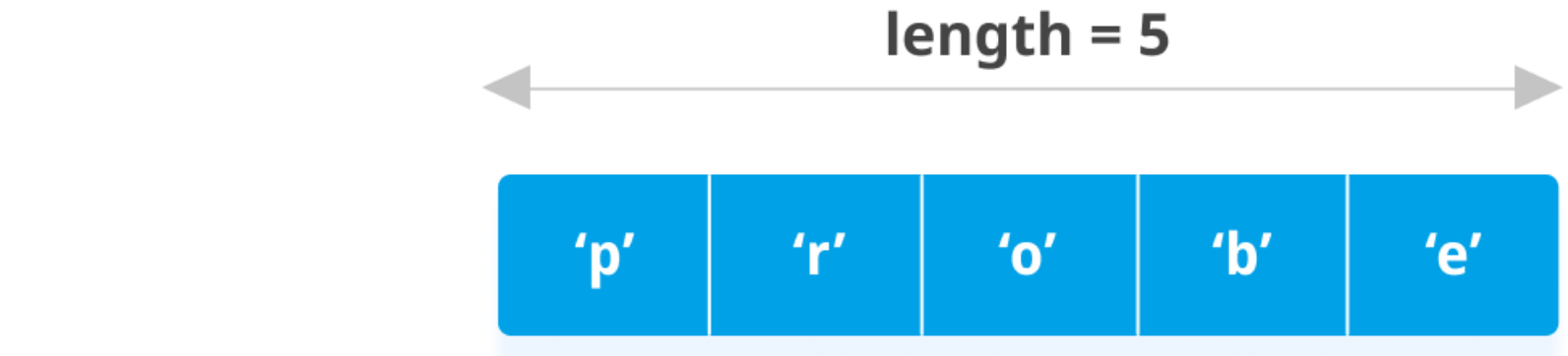
But exclusive at the stop

So you will see **[3]** that was actually the fourth amino acid **p** ***(proline)**. This position was included

But even though **d** ***(aspartic acid)** was at position **5**, the substring does not include that position, but stops at the position before

Hence, `[3:5]` , really grabs everthing from position `[3:4]`

(see exaple of indexing below)



In [20]:
```
# positions start at zero, not one
protein = "vlspadktnv"
print(protein[0:6])
```

```
vlspad
```

In [21]:
```
# if we use a stop position beyond the end, it's the same as using the end
print(protein[0:60])
```

```
vlspadktnv
```

In [22]:
```
# we can also leave the last position blank and it will print until the end
print(protein[0:])
```

```
vlspadktnv
```

In [23]:
```
# we can also leave the first position blank and it will print from the start
print(protein[:5])
```

```
vlspa
```

Can you figure out what will print if you leave both positions blank?

```
print(protein[:])
```

In [25]:
```
# you may also print single characters

protein = "vlspadktnv"
first_residue = protein[0]
```

In [26]:
```
# the positions are also numbered with negative numbers from the back

# So you can either extract the last amino acid two ways

protein = "vlspadktnv"

last_residue = protein[9]
print(last_residue)

last_residue = protein[-1]
print(last_residue)
```

```
v
v
```

In [27]:
```
second_last_residue  = protein[-2]
print(second_last_residue)
```

```
n
```

In [28]:
```
# extract from the back

protein = "vlspadktnv"
print(protein[-4:-2])
```

```
kt
```

In [29]:
```
# What will this print?

'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[4:-1]
```

Out[29]: 'DLVADVAEKTDLSKAKATEVIDAVF'

In [30]:
```
# What will this print?

'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[-5:-4]
```

Out[30]: 'D'

In [31]:
```
# What will this print?

'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[5:5]
```

Out[31]: ''

In [32]:
```
# What will this print?

'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[0:0]
```

Out[32]: ''

In [33]:
```
# What will this print?

'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'[:]
```

Out[33]: 'MNKMDLVADVAEKTDLSKAKATEVIDAVFA'

In [34]:
```
# you can also use the skip function
# the default = 1, meaning no skipping

money = "$5$8$6$7"
money_values = money[1::2]
print(money_values)
```

```
5867
```

In [35]:
```
money = "$5$8$6$7"
money_values = money[-1::-2] #you can also skip backwards. Be careful of your indexing
print(money_values)
```

```
7685
```

In [36]:
```
# how would you write this to create an acronym from the Big Mac Meal (BMM)

order = "TheBigMacMeal"
acronym = order[3:-1:3] # this is not yet correct
print (acronym)
```

```
BMM
```

Counting and finding substrings

In Biology it's very common to ascertain the amount of percentage of something. E.g. the **GC content** of a sequence in a fastq report

We may use the `.count()` method to start this off

In [37]:
```
protein = "vlspadktnv"

# count amino acid residues
valine_count = protein.count('v')
lsp_count = protein.count('lsp')
tryptophan_count = protein.count('w')


# now print the counts
# noticed that I used the three different methods that we saw before
print("valines: " + str(valine_count))
print("tryptophans: %s" % tryptophan_count)
print("lsp:",  lsp_count)
```

```
valines: 2
tryptophans: 0
lsp: 1
```

If you wish to know at which position in your string

Where you first encounter something in you are specifically looking for

Then you use the `.find()` method

In [38]:
```
protein = "vlspadktnv"

print(str(protein.find('p')))
print(str(protein.find('kt')))
print(str(protein.find('w'))) # .find() gives "-1" as output if what you are looking for is not in the strir
```

```
3
6
-1
```

String operators:

```
in and not in
```

In [39]:
```
TATA_box = 'TATATATATATATATATATATATA'

'TATA' in TATA_box

# gives a Boolean as output: True or False
```

Out[39]: True

In [40]:
```
"GC" in TATA_box
```

Out[40]: False

In [41]:
```
"GC" not in TATA_box
```

Out[41]: True

In [*]:
```
# Let's play with some code

your_dna = input("\n\nPlease Enter your DNA to see whether you are a superhuman: \n\n"
                 "If your name starts with A - G, your DNA is GATCA\n"
                 "If your name starts with H - N, your DNA is CATT\n"
                 "If your name starts with O - S, your DNA is AGAT\n"
                 "If your name starts with T - Z, your DNA is TAAT\n")

your_dna = your_dna.upper()


superhuman_DNA = "ATTCCATCAAGCTGATCAGGTTATCCATCTAGATCATNNATAAAGTACTGGGCATGCAA"
if your_dna in superhuman_DNA:
    print ("\nCongratulations. Your DNA shows that you have superpowers!!")

elif your_dna == "CATT" or  your_dna == "TAAT":
    print("\nI'm sorry. Your DNA shows that you are NAUGHTY.\n"
       "Maybe try to be on Santa's good list next year.")

else:
    print("\n\nERROR: Please enter the correct DNA sequence")
```

```
Please Enter your DNA to see whether you are a superhuman:

If your name starts with A - G, your DNA is GATCA
If your name starts with H - N, your DNA is CATT
If your name starts with O - S, your DNA is AGAT
If your name starts with T - Z, your DNA is TAAT
```

Use `help()` to find all the methods you can use when you are working with strings

You can try it now to see the methods we have already used

> And which may be useful to you

In [*]:
```
help(str)
```

EXERCISES

1. Calculating AT content

Here's a short DNA sequence: ACTGATCGATTACGTATAGTATTTGCTATCATACATATATATCGATGCGTTCAT

Write a program that will print out the AT content of this DNA sequence. Hint: you can use normal mathematical symbols like add (+), subtract (-), multiply (*), divide (/) and parentheses to carry out calculations on numbers in Python.

In [23]:
```
my_sequence = "ACTGATCGATTACGTATAGTATTTGCTATCATACATATATATCGATGCGTTCAT"

count_As = my_sequence.count("A")
count_Ts = my_sequence.count("T")
#AT_count = count_As + count_Ts
length_seq = len(my_sequence)

AT_perc = ((count_As + count_Ts)/length_seq)*100

print (AT_perc)
```

```
68.51851851851852
```

2. Complementing DNA

Here's a short DNA sequence: ACTGATCGATTACGTATAGTATTTGCTATCATACATATATATCGATGCGTTCAT

Write a program that will print the complement of this sequence.

In [24]:
```
# replace them all with small letters,
# so that they don't keep replacing the nucleotides that were already replaced

my_sequence = "ACTGATCGATTACGTATAGTATTTGCTATCATACATATATATCGATGCGTTCAT"

replacement_1 = my_sequence.replace("A", "t")

replacement_2 = replacement_1.replace("C", "g")

replacement_3 = replacement_2.replace("T", "a")

replacement_4 = replacement_3.replace("G", "c")

print(replacement_4)
my_compl = replacement_4.upper()

print(my_sequence)
print(my_compl)
```

```
tgactagctaatgcatatcataaacgatagtatgtatatatagctacgcaagta
ACTGATCGATTACGTATAGTATTTGCTATCATACATATATATCGATGCGTTCAT
TGACTAGCTAATGCATATCATAAACGATAGTATGTATATATAGCTACGCAAGTA
```

3. Restriction fragment lengths

Here's a short DNA sequence: ACTGATCGATTACGTATAGTAGAATTCTATCATACATATATATCGATGCGTTCAT

The sequence contains a recognition site for the EcoRI restriction enzyme, which cuts at the motif G*AATTC (the position of the cut is indicated by an asterisk). Write a program which will calculate the size of the two fragments that will be produced when the DNA sequence is digested with EcoRI.

In [25]:
```
# I like to first start with a tester

# What are my steps?:
# 1. I need to be able to find this sequence,
#    and its position in the dna:                              .find()
# 2. In order to know the length of 2nd fragments,
#    I need to know the length of the entire DNA sequence      .len()

test_seq = "TAGAATTCTA"
position_1st_fragm = test_seq.find("GAATTC")
print(position_1st_fragm)
```

```
2
```

In [26]:
```
# note that becaue the positions start at 0,
# you have to add "1" to make sure the math is correct
len_1st_fragment = position_1st_fragm + 1
print(len_1st_fragment)
```

```
3
```

In [27]:
```
len_test_seq = len(test_seq)
print(len_test_seq)


len_2nd_fragment = len_test_seq - len_1st_fragment
print(len_2nd_fragment)

# Now that I know this is working, I just apply it to my real sequence
```

```
10
7
```

In [28]:
```
my_sequence = "ACTGATCGATTACGTATAGTAGAATTCTATCATACATATATATCGATGCGTTCAT"

position_1st_fragm = my_sequence.find("GAATTC")
len_1st_fragment = position_1st_fragm + 1
print(len_1st_fragment)

length_my_sequence = len(my_sequence)
print(length_my_sequence)

len_2nd_fragment = length_my_sequence - len_1st_fragment
print(len_2nd_fragment)
```

```
22
55
33
```

4. Splicing out introns, part one

Here's a short section of genomic DNA:
ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCGATCGATCGATCGATCGATCATGCTATCATCG

It comprises two exons and an intron. The first exon runs from the start of the sequence to the sixty-third character, and the second exon runs from the ninety-first character to the end of the sequence. Write a program that will print just the coding regions of the DNA sequence.

In [29]:
```
# to make sure that I don't get confused about the positions of the characters
# I'm going to create a test sequence
# If this question had said that the first exon was from the start to the 4th character,
# what would that look like?
# Make sure that you understand whether they mean the 4th character from a coder's view
# or from a non-programmer's view
# Here they mean up until, but NOT INCLUDING the 4th character (coder's view)


test_seq = "ATCGAAAACTTT"

# length_test_seq = len(test_seq)
# print(length_test_seq)

# So how will I slice from the start and UP to and EXCLUDING character 4?
# this means, we need to stop at the first 'C'

# so now I already know that I'm going to use "62" in the real problem
test_first_exon = test_seq[:3]
print(test_first_exon)
```

```
ATC
```

In [30]:
```

# so if the second exon ran from the 9th position to the end, how will I slice that?
# it looks like the last "C" is the 9th character, and it lies at position "8"
test_last_exon = test_seq[8:] # so now I know that I will have to use the "90th" position in the real proble
print(test_last_exon)
```

```
CTTT
```

In [31]:
```
my_dna = "ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCGATCGATCGATCGATCATGCTATCATCGA

# Again, the person mentions up until, but not including, the 63rd character
# They mean that they only want 62 characters.
# At least this is how a non-programmer would say it
# Since indexing starts at 0
# 0 to 61 will be 62 characters
# Don't be alarmed if you got confused here.
# This is just an example of how you need to make sure.....
# That you and the other person are on the same page

first_exon = my_dna[:62] # this is index 0 - 61
#print (len(first_exon))
last_exon = my_dna[90:] # this is index 90 - infinity
#print (len(last_exon))

#finally I must concatenate the exons

combined_exons = first_exon + last_exon

print(combined_exons)
print(len(combined_exons))
```

```
ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGATCATCGATCGATATCGATGCATCACTACTAT
94
```

5. Splicing out introns, part two

Using the data from part one, write a program that will calculate what percentage of the DNA sequence is coding.

In [32]:
```
percent_CDS = (len(combined_exons)/length_my_dna)*100
print (round(percent_CDS))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-32-6cf146e2ff92> in <module>
----> 1 percent_CDS = (len(combined_exons)/length_my_dna)*100
      2 print (round(percent_CDS))

NameError: name 'length_my_dna' is not defined
```

6. Splicing out introns, part three

Using the data from part one, write a program that will print out the original genomic DNA sequence with coding bases in uppercase and non-coding bases in lowercase.

In [41]:
```
# let's go back to testing with my test sequence first


intron = (test_seq[4:8]).lower()
print(intron)

comb_exons_introns = test_first_exon + intron + test_last_exon
print(comb_exons_introns)


# now I can apply it to mine
```

```
aaaa
ATCaaaaCTTT
```

In [42]:
```
intron = (my_dna[62:90]).lower() # this is from index/position 62 - 89
modified_dna = first_exon + intron + last_exon

print(modified_dna)
```

```
ATCGATCGATCGATCGACTGACTAGTCATAGCTATGCATGTAGCTACTCGATCGATCGATCGatcgatcgatcgatcgatcgatcatgctATCATCGATCGATATCG
ATGCATCACTACTAT
```

BioPython

We can use BioPython to do some of the tasks that we did manually

Find the **[Link to the BioPython tutorial and cookbook below]** [http://biopython.org/DIST/docs/tutorial/Tutorial.html](http://biopython.org/DIST/docs/tutorial/Tutorial.html%5D)

In [33]:
```
# There is a BioPython library for calculating GC content..
# Since it's pretty common task for Biologist

from Bio.Seq import Seq
from Bio.SeqUtils import GC
my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC")
GC(my_seq)
```

Out[33]: 46.875

In [34]:
```
# Get the comleent of a DNA seqeunce

from Bio.Seq import Seq
my_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC")
my_seq.complement()
```

Out[34]: Seq('CTAGCTACCCGGATATATCCTAGCTTTTAGCG')

In [35]:
```
my_seq.reverse_complement()
```

Out[35]: Seq('GCGATTTTCGATCCTATATAGGCCCATCGATC')

In [36]:
```
# Reverse complement coding sequences (CDS)

from Bio.Seq import Seq
coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG")
template_dna = coding_dna.reverse_complement()
template_dna
```

Out[36]:
```
Seq('CTATCGGGCACCCTTTCAGCGGCCCATTACAATGGCCAT')
```

In [37]:
```
# Transcribe CDS
messenger_rna = coding_dna.transcribe()
messenger_rna
```

Out[37]:
```
Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG')
```

In [38]:
```
# Translate mRNA

messenger_rna = Seq("AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG")
translated_mRNA = messenger_rna.translate()
translated_mRNA
```

Out[38]:
```
Seq('MAIVMGR*KGAR*')
```

In [39]:
```
# some translations are unresolved
# So you might need to specify the use of the correct Translation table

translated_mRNA = coding_dna.translate(table="Vertebrate Mitochondrial")
translated_mRNA
```

Out[39]:
```
Seq('MAIVMGRWKGAR*')
```

In [40]:
```
# Now, you may want to translate the nucleotides
# up to the first in frame stop codon
# and then stop (as happens in nature)

translated_mRNA =  coding_dna.translate(table =  "Vertebrate Mitochondrial", to_stop=True)
translated_mRNA
```

Out[40]:
```
Seq('MAIVMGRWKGAR')
```

THE END