

Sequence data manipulation

Practical 2

Contents

| | |
|---|----|
| Task 1: Storing a short DNA sequence | 2 |
| Plain Python script | 4 |
| Linting..... | 7 |
| Debugging | 7 |
| Jupyter Notebook | 8 |
| Task 2: Concatenating DNA sequences..... | 10 |
| Task 3: Generating random DNA sequence..... | 11 |
| Task 4: Mutating DNA | 12 |
| Task 5: Making complementary DNA..... | 13 |

During the Practical 1 you should have learned how to write Python scripts in VS Code, how to switch between Python environments, and even created some simple Python projects.

In most of the remaining practicals, you will not need to install any additional packages. So, you should not worry about what Python interpreter you use later today: any of them should work well.

Understandably, most of our tasks during the practical sessions will be about using Python in *bioinformatics*. Also, the practicals will implement what we have learnt previously in the lectures. This practical session will be about manipulating strings, which was introduced in Lecture 3 that you attended this morning. To be exact, we will use basic Python commands to manipulate strings representing genes and proteins sequences.

Genetic sequences could be treated by Python like any other strings of characters: at the end of the day, any combination of ATCG-s is just a bunch of characters. Python doesn't care what the characters mean :) In contrast, these characters do mean something to biologists. The meaning of ATCG-s (and some other symbols) is summarised in Tables 1 and 2: **Table 1** presents the standard coding of nucleic acids and **Table 2** presents the standard coding of amino acids that we are going to use throughout the Python module. Have a look at them if you didn't work with these codes before.

| Code | Nucleic Acid(s) | Code | Nucleic Acid(s) |
|------|-----------------|------|------------------------|
| A | Adenine | S | C or G (strong) |
| C | Cytosine | Y | C or T (pyrimidine) |
| G | Guanine | K | G or T (keto) |
| T | Thymine | V | A or C or G |
| U | Uracil | H | A or C or T |
| M | A or C (amino) | D | A or G or T |
| R | A or G (purine) | B | C or G or T |
| W | A or T (weak) | N | A or G or C or T (any) |

Table 1. Standard IUB/IUPAC nucleic acid codes

Task 1: Storing a short DNA sequence

Let's assume that we have found an interesting DNA sequence in a database (or got it from a colleague). The first step is to store the sequence in Python, so we can do something with it later.

Here is the short DNA sequence that we will use in this practical session:

```
String 1: ACGGTAGCTAGTTTCGACTGGAGGGGTA
```

As you may remember, we can write and run Python code in several ways. During this practical, first, you will write a plain Python script. Then, you will repeat it in the Jupyter Notebook style. In both cases, you will use VS Code as your IDE.

| Code | Amino Acid(s) | Alternate Code |
|------|-----------------------------|----------------|
| A | Alanine | Ala |
| B | Aspartic acid or Asparagine | Asx |
| C | Cysteine | Cys |
| D | Aspartic acid | Asp |
| E | Glutamic acid | Glu |
| F | Phenylalanine | Phe |
| G | Glycine | Gly |
| H | Histidine | His |
| I | Isoleucine | Ile |
| K | Lysine | Lys |
| L | Leucine | Leu |
| M | Methionine | Met |
| N | Asparagine | Asn |
| P | Proline | Pro |
| Q | Glutamine | Gln |
| R | Arginine | Arg |
| S | Serine | Ser |
| T | Threonine | Thr |
| V | Valine | Val |
| W | Tryptophan | Trp |
| X | Unknown | Xxx |
| Y | Tyrosine | Tyr |
| Z | Glutamic acid or Glutamine | Glx |

Table 2. Standard IUB/IUPAC amino acid codes.

Plain Python script

Open VS Code and make a new project folder called **practical_02**. Keep all your scripts for this practical session within this project folder. Because we will only use in-built Python features, you should not worry about what Python interpreter and environment to use: no need in any additional packages for now :)

Create a new file, give it a meaningful name, e.g. **dna_sequence.py** (do not use spaces in the file names!).

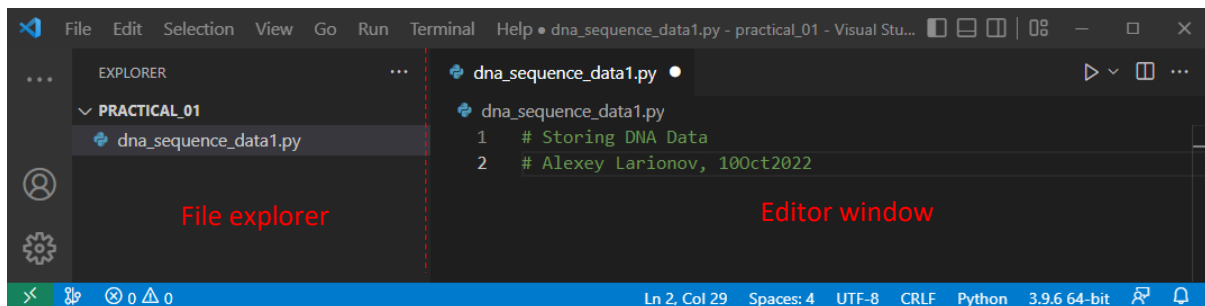
Next, we should provide a comment to tell other people (and ourselves – when you revisit the program in three years' time) what the program is going to do:

```
1: # Storing DNA data
```

and who and when wrote the file:

```
2: # Your Name, Date
```

At this point your screen may look like this:



Then, we need to save our DNA sequence to a Python variable.

Q: What is the most appropriate variable name?

my_variable ?

dna ?

amino_acid ?

Use your answer to save your DNA sequence to a variable:

```
3: ??? = 'ACGGTAGCTAGTTTCGACTGGAGGGGTA'
```

Use single quote here.

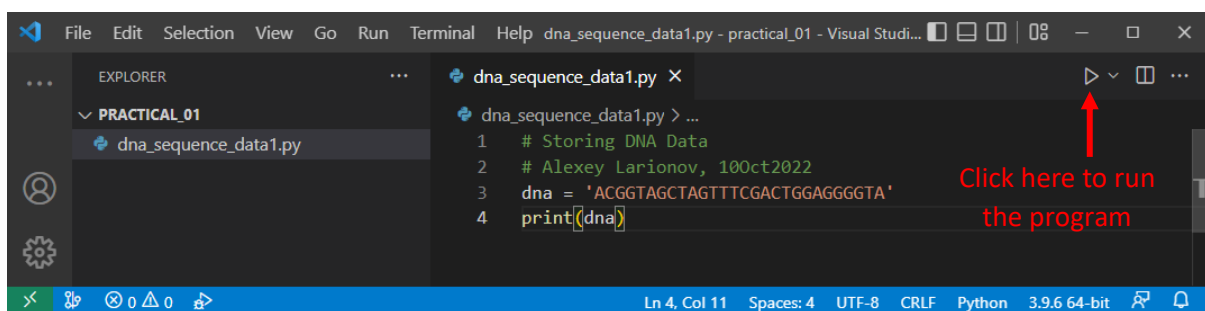
Finally, we want to print out our data to the screen when we run the program.

Again, using your answer from the last question:

```
4: print(????)
```

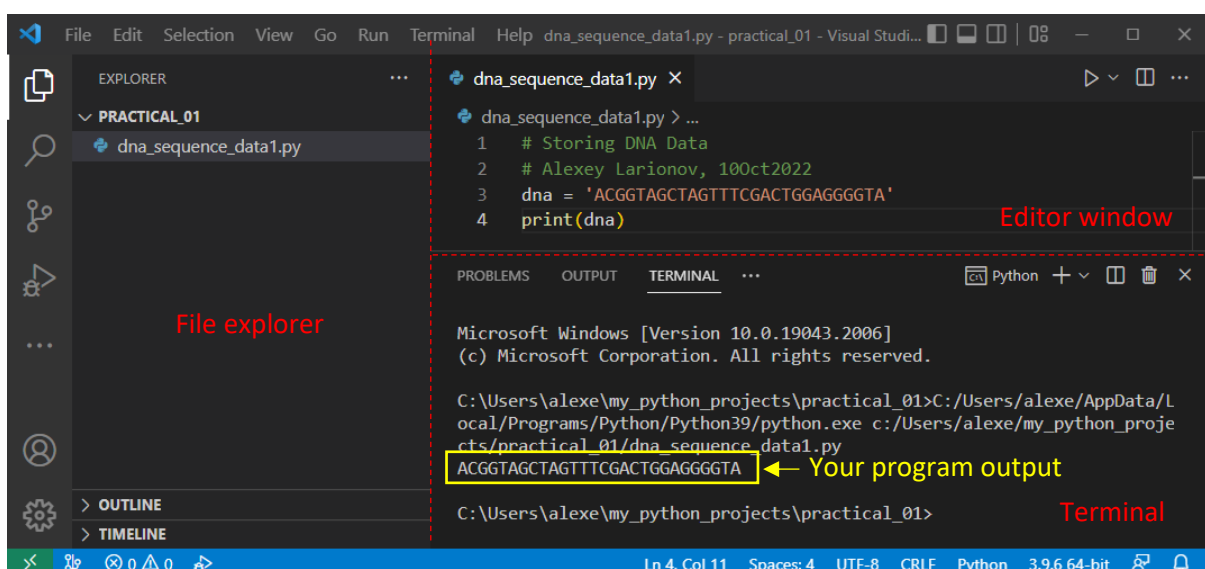
Obviously, the program needs to be saved. Use File > Save or Ctrl+S.

Note on the screenshot below that **dot** near the file name in the header of editor window has changed to **cross** (X) : this indicates that the file has been saved.



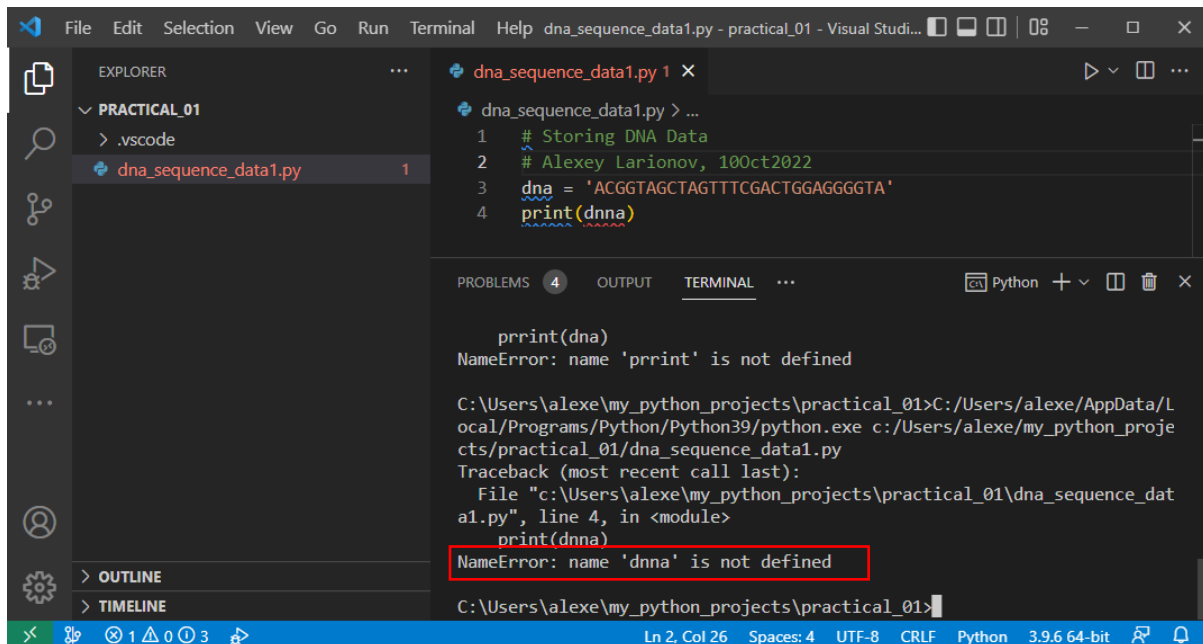
Once the program has been saved, you should be in a position to run it: click to the arrow as shown above.

Output of the successfully completed program will be shown in Terminal, which should look like this:



However, sometime things may go wrong ... You may encounter **ERROS !**

If this happens, Python provides the error(s) description. Usually, you may find the main cause of the error at the end of the description (I encircled it in red here):



```
File Edit Selection View Go Run Terminal Help dna_sequence_data1.py - practical_01 - Visual Studi...
EXPLORER
PRACTICAL_01
> .vscode
+ dna_sequence_data1.py 1
dna_sequence_data1.py 1
1 # Storing DNA Data
2 # Alexey Larionov, 10Oct2022
3 dna = 'ACGGTAGCTAGTTTCGACTGGAGGGTA'
4 print(dnna)

PROBLEMS 4 OUTPUT TERMINAL ...
Python + - 
prnt(dna)
NameError: name 'prnt' is not defined

C:\Users\alexe\my_python_projects\practical_01>C:/Users/alexe/AppData/L
ocal/Programs/Python/Python39/python.exe c:/Users/alexe/my_python_proje
cts/practical_01/dna_sequence_data1.py
Traceback (most recent call last):
  File "c:\Users\alexe\my_python_projects\practical_01\dna_sequence_dat
a1.py", line 4, in <module>
    print(dnna)
NameError: name 'dnna' is not defined

C:\Users\alexe\my_python_projects\practical_01>
```

Q: What caused the error in the above figure?

Hint: Obviously, Python has a problem interpreting what you have written.

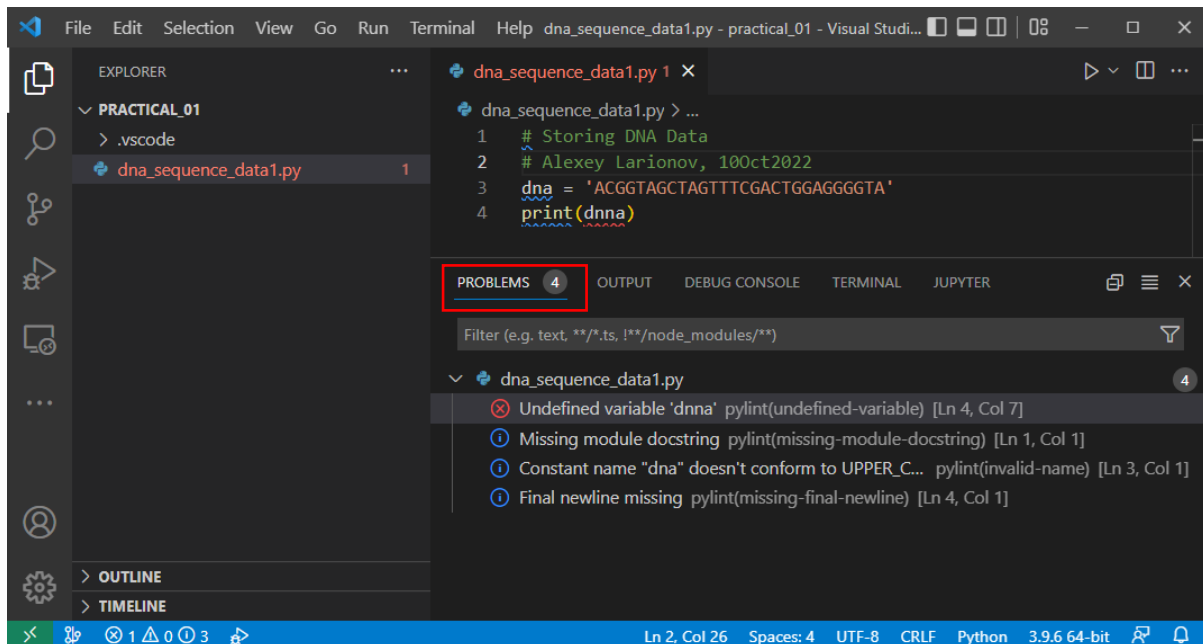
Have comments been prefixed with a # (hash) ?

Have I properly enclosed the string with quotes ?

Have I made any typos ?

Linting

If your VS Code has properly configured and activated **linting**, it may spot some syntax errors even before you run the code. **Linting** is a set of the IDE's features, which help to clean the code (https://en.wikipedia.org/wiki/Lint_remover). In the screenshot below the errors are underlined in red, style suggestions are shown with blue, and the detailed description of the detected problems is given in the Problems tab of the VS Code multifunctional terminal:

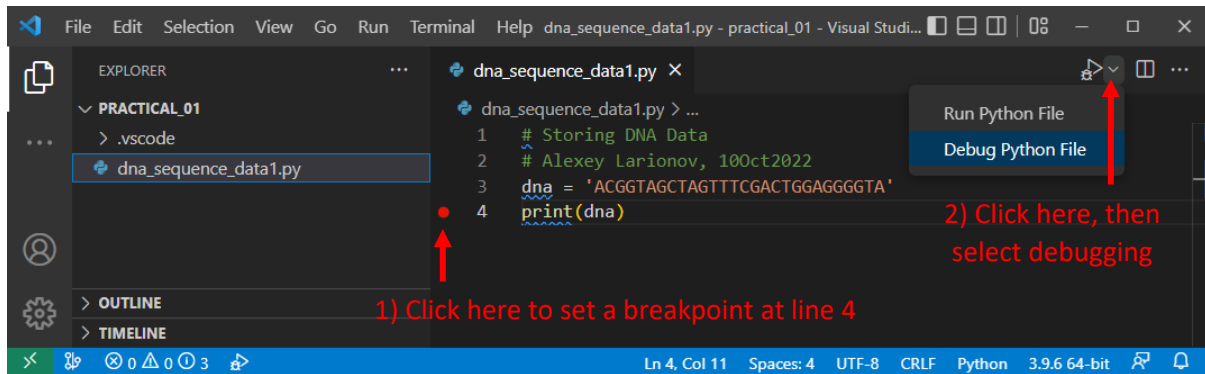


If you don't see linting in your code, you may need to activate it or install a linting extension (e.g. **Pylance** or **Pylint**). Usually, **Pylance** should be automatically installed with Python extension in VS Code. Don't worry about adding linting package(s) to your **global** environment (or to **base** Conda environment): **Pylance** and **Pylint** are well written and very unlikely to cause conflicts.

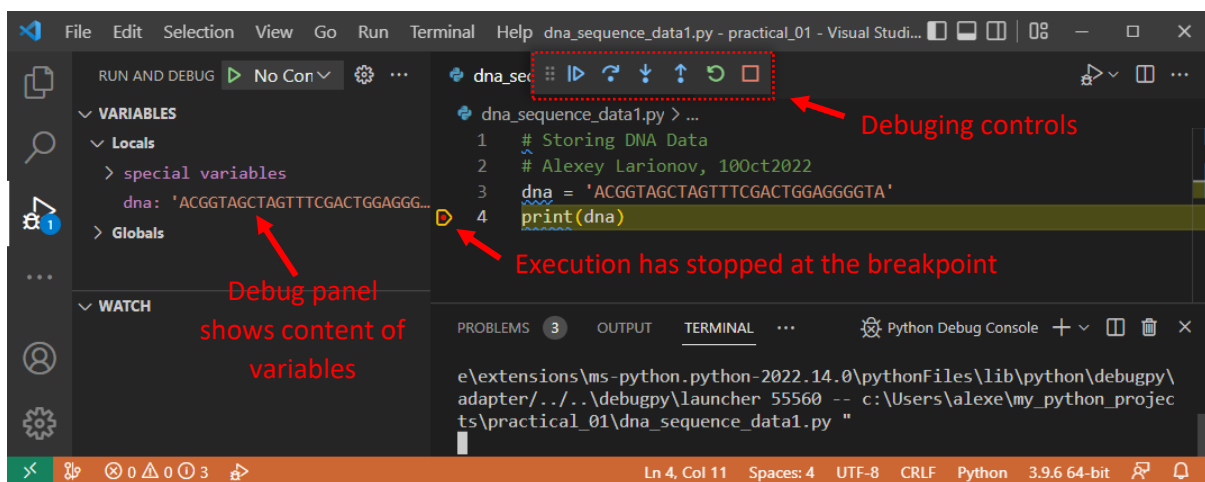
Debugging

Finding and removing errors in programs is called **debugging**. It's easy to spot simple error(s) in a short script. It's much harder to find errors in a complex program with many variables and functions. In such case, debugging requires running program step-by step and inspecting variables after each step. The opportunity for running the program in the **debug mode** is provided in VS Code (and in any other good IDE).


To run your script in the debug mode in VS Code, first you need to set a break-point in the script: to indicate from what step to start debugging. Click at the left of the required code line to put a breakpoint: it will be shown as a red dot (see figure on the next page). Then click at the arrow in top right corner of Editor window and select the "Debug Python File" option (again, as shown in figure on the next page).



If you successfully set the breakpoint and started debugging, you should see screen like this:



Note that the status bar became orange, instead of blue; the left panel shows debug information, including the variables explorer; and the debugging controls appear at the top of the screen.

Press  control to go to the next step of your code (in the shown example it will complete the code execution because line 4 is the last line of the script). To remove a breakpoint just click on it again.

In this exercise you only tried the basic features available for code linting and debugging. There are more features in VS Code to detect and remove errors in a program, and to improve the style of your coding.

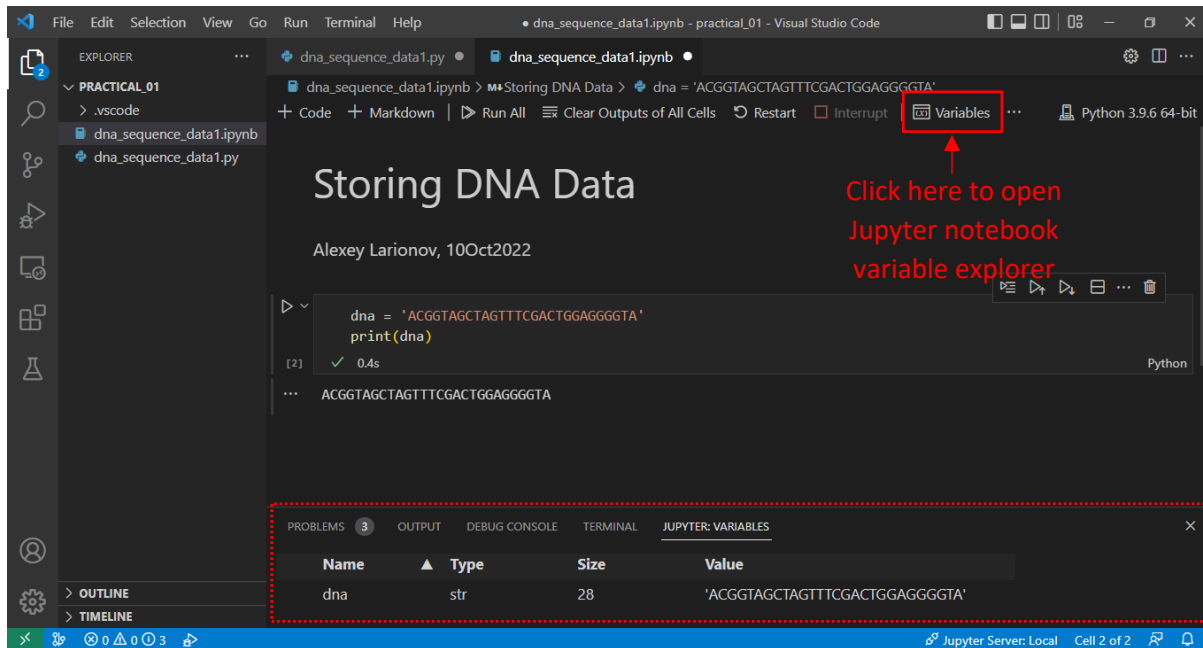
Jupyter Notebook

Now, execute the same code using VS Code Jupyter Notebook. Briefly:

- Add a new file to the project folder, call it `dna_sequence.ipynb`
- Put required lines of code in the code cell
- Put comments in a markup cell before code
- Render markup cell and execute code cell (agree to install `ipkernel` package into your environment if it was not installed before)
- Save notebook and export results to HTML log.

If needed, refer to the detailed instructions given in Part 3 of Practical 1.

A successfully executed Jupyter Notebook may look like shown in the figure below. The figure also illustrates the Jupyter Notebook variable explorer.



Now, when you practiced using VS Code to write and execute Python programs as plain scripts and as Jupyter Notebooks, you may choose by yourself what style to use for the remaining exercises in this practical session.

Exercise 2.1. Try assigning other characters and numbers combinations to another variable and printing it out. Do they work as expected?

Exercise 2.2. Repeat the last exercise "`dna_sequence.py/ipynb`" using double-quotes (") instead of single quotes ('). Are the results the same? Why?

Exercise 2.3. Deliberately introduce errors into your code (e.g. remove # in front of comments, remove a quotation mark, make a typo, etc). Investigate how Python responds to these errors. Try to run "`Hello_error_world.py`" (provided in `scripts.zip`), find and remove the errors one by one.

Keep your workplace organised! Make a separate script/ notebook for each exercise. Keep all scripts/notebooks inside the `practical_2` project folder.

Task 2: Concatenating DNA sequences

All we did so far was putting some DNA sequence into a Python variable. It's time now to start doing something more with it !

In a biology lab, it's pretty common to join pieces of DNA together. Let's write a computational equivalent of it. For this you need another piece of DNA sequence:

```
DNA 2 string:  GACATTTTCAGATTGA
```

Now you wish to add it to the string **DNA 1** string that you already used before. Although you could just modify your existing code, please write code in a new file: so that you can revisit the processes and build up your experience in Python programming.

Make a new Python script/ notebook, call it *concatenating_dna.py* or *concatenating_dna.ipynb*

Having done the Task 1, you may already guess the initial lines of your new code, which may look something like this (if you write it as a plain script):

```
1:  # Concatenating DNA fragments
2:  # Your name and date
```

Getting two sequences of DNA into your Python program shouldn't be a problem. You may put them into two separate variables:

```
3:  dna1 = 'ACGGTAGCTAGTTTCGACTGGAGGGGTA'
4:  dna2 = 'GACATTTTCAGATTGA'
```

To make sure that our variables really contain our DNA sequences as expected, let's print them out:

```
5:  print("First DNA sequence: ", dna1)
6:  print("Second DNA sequence: ", dna2)
```

By the way, although I showed the line numbers above, it's a good practice to add empty lines and comments to your code, to improve the readability. If you prefer using Jupyter Notebook, then it's a common practice to put different bits of code into different cells, and add markdown cells between for comments. In this case the line numbers may become irrelevant. So, I will not include the line numbers any longer.

OK, now you have the DNA sequences in variables, and you can start joining them together (concatenating). Python has several ways for concatenating strings. Try these three methods:

Method 1: Using "+" operator

```
# String concatenation using "+" operator
dna3 = dna1 + dna2
print("Concatenated DNA sequence [Method 1]: ", dna3)
```

Method 2: Using `"".join()` function

```
# String concatenation using "".join() function
dna4 = "".join([dna1,dna2])
print ("Concatenated DNA sequence [Method 2]: ", dna4)
```

Method 3: Using `%` string operator

```
# String concatenation using % operator
dna5 = "%s%s" % (dna1, dna2) # %s means format as string
print("Concatenated DNA sequence [Method 3]: ", dna5)
```

Save your program and run it. Which method would you prefer for concatenating two short DNA sequences ?

Exercise 2.4. Investigate each concatenation method. What happens if you try them with numbers?

Exercise 2.5. Try the string repetition operator (`*`) on numbers and strings. What are the results? Say, the number = 5 or 3.142, and string = "Hello, world!"

Task 3: Generating random DNA sequence

Now, rather than using DNA sequences given to you by others, you will make your very own random DNA sequence! For this, add a new file to the project, and call it **`random_dna.py`** or **`random_dna.ipynb`**. Note that it's important to give the files meaningful names.

In this part of the practical session, you will work with DNA sequences using two different Python data types: strings and lists. You will learn more about lists tomorrow. For now, just think about a list as a collection of objects. Of course, conceptually, a list of letters is very similar to a string! However, from a technical perspective, Python processes strings and lists differently, using different functions and operators.

You already know that Python comes with many useful functions stored in separate libraries/ packages/ modules. For this task we will use a Python library called **`random`**. This is an in-built library, so you don't need to install it with PIP or Conda. This library implements procedures to generate (pseudo)random numbers, and other similar tasks (like selecting random items from a collection).

For making a sequence of DNA nucleotides, we will use a function called `random.choices()`.

However, first we need to import the library:

```
import random
```

Now we need to decide what length of the DNA sequence we want to generate; let's set it to 100 bases:

```
N = 100 # Number of bases
```

Finally, we may generate the sequence:

```
alphabet = ['A', 'T', 'G', 'C'] # Four DNA nucleotides
sequence = random.choices(alphabet, k=N)
```

This code picks bases randomly from the four bases (`alphabet`) and saves 100 randomly chosen bases in a list called `sequence`.

Usually, DNA sequences are kept as strings. So, we will use `''.join()` function to convert our list of DNA bases to a DNA string:

```
random_dna = ''.join(sequence)
print(sequence)
print(random_dna)
```

Save and execute your program at this point. Debug, if necessary.

Hurray! You have made your very own (random) DNA sequence !

Task 4: Mutating DNA

Now, let's mutate the DNA that we just created !

For introducing a single point mutation at a random position in our DNA sequence we will use functions `randint()` and `random.choice()`. Note that function `random.choice()` is different from the `random.choices()` that we used previously.

`randint()` will generate a random integer number between 0 and 99 that will be the position for our mutation:

```
mutation_site = random.randint(0, len(sequence) - 1)
```

Now, we assign a random nucleotide to the previously selected position:

```
sequence[mutation_site] = random.choice(alphabet)
```

Again, convert the list containing mutated DNA sequence into the DNA string:

```
mutated_DNA = ''.join(sequence)
print(mutated_dna)
```

Save and execute the whole program.

Q: What python data type is the variable called "alphabet" ?
Q: Why we subtract one in "len(sequence) - 1" in the code?
Q: What happens if you run the program again?
Q: Why occasionally the mutation doesn't happen?
Q: Can you show (e.g. underline or colour) the mutated base?
(optional: only for students with prior Python experience!)

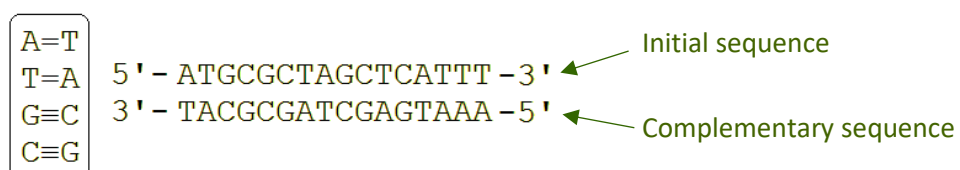
Exercise 2.6 (optional). Introduce 1% mutations into a randomly generated DNA sequence of length 500 bp.

Task 5: Making complementary DNA

Let's make the complementary sequence to this piece of DNA:

ACTGATCGATTACGTATAGTATTTGCTATCATAATATATATCGATGCGTTCAT

DNA sequence has two complementary base pairs: the **G≡C** pair (with three hydrogen bonds) and the **A=T** pair (with two hydrogen bonds). So, each DNA sequence has a complementary sequence, as illustrated below:



At the first glance, our task seems pretty straightforward. Make a new script or Jupyter notebook, call it *complementary_dna.py* / *ipynb*. Then use function `str.replace()` for replacing letters in strings:

```
'A' with 'T'  
'T' with 'A'  
'C' with 'G'  
'G' with 'C'
```

The code below implements this idea, using output of one `str.replace()` call as the input for the next:

```
dna = "ACTGATCGATTACGTATAGTATTTGCTATCATACATATATATCGATGCGTTCAT"  
  
# Replace A with T  
replacement1 = dna.replace ('A', 'T')  
  
# Replace T with A  
replacement2 = replacement1.replace ('T', 'A')  
  
# Replace C with G  
replacement3 = replacement2.replace ('C', 'G')  
  
# Replace G with C  
replacement4 = replacement3.replace ('G', 'C')  
  
# Print the result  
print(replacement4)
```

The output:

```
ACACAACCAAAACCAAAACAAAAACCAAACAAACAAAAAAACCAACCCAACAA
```

Q: Does the result look right?

Exercise 2.7. Identify the problem with the above solution and fix it.

Well done! You have finished Practical 2