1 Linux basics

In the linux operating system, plenty tasks can be accomplished from the terminal without of a graphic user interface. Files can be manipulated using basic commands, such as `cd` for changing directories, `cp` for copying files, `gedit`, `vi` or `nano` for editing text files, `grep` for searching through files for- and printing the keyword. Information on these commands can be found by bringing up the manual with the `man` command. For example, in the manual for the `ls` command, we see that its function is listing the contents of the directory the terminal is running in.

Arguments can also be applied to the commands, altering the command. To name a few, for the `ls` command, `-a` argument lists all files including hidden ones, and `-R` lists all subdirectories recursively, listing all the containing files. The `wc` command prints the newline, word and byte counts for the selected file, while the argument `-l` prints only the number of lines. With the argument `-n` of `sort`, the files are sorted in a numeric instead of alphabetical order. In the `grep` command, the arguments `-i` ignores case distinction, `-v` inverts the match, selecting non-matching lines and `-c` counts the number of found keywords.

Plenty commands such as grep, cat, head, tail, sort, wc, etc, perform an action on a selected file. A file must therefore be specified when calling the command, else the action will be stuck, trying to complete its function without a target. In such instances, the command will keep running indefinitely and the terminal will be unresponsive. The command can be terminated by pressing the control and C keys simultaneously.

Wildcards represent various characters - * represents one or more characters, ? any single characters and [] a character written within the brackets. With these, the `cat` command can collect the contents of files starting with `seq` with a total of 5 characters with `seq??`, while the output of the command can be piped with > to a new file. `ls -1` lists the files in the directory, one per line, while `| wc -l` counts the lines, returning the number of files.

Using the `head` and `tail` commands, the first 10 lines can be viewed, or a specific number by including it as an argument. Specific instances of text can be found by the `grep` command, while `diff` displays the differences between files and returns nothing if the files are identical.

Piping can shorten our script into only one command. Thus, searching for > using `grep` and piping the word count of lines command can produce the number of sequences in a FASTA file -44 – while inverting the grep with `-v` produces the number of lines of sequence – 473. Similarly, using grep to search for 'LOCUS' in testdata.gb while piping it to `wc -l` gives the number of entries listed – 24, and searching for 'ORGANISM' and piping it to `sort` provides a list of all organisms in alphabetical order, of which *Pasteuria penetrans* is the most prevalent.

A collection of sequencing files was investigated in /export/data/657.019/frag/ using the text processing commands, piping and automation. A total of 13337 fragments were found, with 425 containing read errors and 12912 without. A total of 13324 unique fragments were found using the `uniq` command. Ignoring the case decreased the number to 13318, while further ignoring the fragments with read errors dropped the number to 12892.

2 Bash scripting

2.1 Echoing arguments

In the script 2.1_echoargs.sh, a filename variable was established calling the basename of the script and was echoed. A total number of arguments were printed using `$#`, while `$@` was used to print all arguments. The script was run in the terminal using zero, one and multiple arguments.

2.2 Copying files given as arguments

To create backups, the script 2.2_backup.sh was written. First, an if loop was established, echoing the message that the script needs at least one file in the case the number of arguments was less than 1, else it confirmed having at least one file. The total number of files was relayed in another message. Backup files were copied with a for loop, iterating over all arguments, copying them to the backup folder and appending `.bak` to the original name.

2.3 Printing only some lines of a file

The 2.3_firstlast.sh script was written to print the first and the last line of a text file given in an argument. First, an if loop was established to, in case of the number of arguments being less than 1, inform the user that it needs at least one file. another if loop was created with the parameter to check if the file is readable and informing the user of the success, with the negative confirmation under the else clause. A variable was created to count the lines in the file using the cat command and piping it to `wc -l`, with the output being the number of lines. An if loop was established to check if the count of the lines was less than two and informing the user, else echoing the confirmation. The first line was printed using the `head` command with the argument `-1` to only print one line. The last line was extracted using `tail` in a similar manner.

2.4 Adding columns to a text file

The purpose of the 2.4_nlcw.sh script is to list the lines in a file while also counting the number of words in a file. An if loop was established, if the number of arguments is less than one, then the script prompts the user to enter a file and uses it for the analysis, else the first argument is considered as the investigated file. A header line was printed, listing first the line number, then word count and lastly the contents of the line, separating the elements with tabs. A line counter was initialized. A while loop was used to read the file one line at a time, each time increasing the line counter. A variable was established, piping the echoed line to a word count command. For each line, the line number, the word count and the content of the line were printed, separated by tabs.

3 R

3.1 Mietspiegel München (1994) survey

The survey in question – “Mietspiegel München (1994)” – relays data on rental properties in München in the year 1994. The survey was given in a tab separated format, containing the survey title in the first row, with the variables named in the second row and data given in subsequent rows. Thus, the first row was omitted while importing the data. The acquired dataset contained 1082 entries. The subjects of the survey can be found in Table 1.

Table 1 Subjects of the survey "Mietspiegel München(1994) "

nm	Net rent in DM
wfl	Floor space in m^2
bj	Year of construction
bad0	Bathroom in apartment? (1: no; 0: yes)

zh	Central heating? (1: yes; 0: no)
ww0	Hot water supply? (1: no; 0: yes)
badkach	Tiled bathroom? (1: yes; 0: no)
fenster	Window type (1: plain windows; 0: state-of-the-art windows)
kueche	Kitchen type (1: well equipped kitchen; 0: plain kitchen)
mvdauer	Lease duration in years
bjkat	Age category (1: to 1918; 2: 1919 to 1948; 3: 1949 to 1965; 4: 1966 to 977; 5: 1978 to 1983; 6: 1984 onwards
wflkat	Floor space categories (1: up to 50 m^2; 2: 51 m^2 to 80 m^2; 3: larger than 80 m^2
nmqm	Net rent per sqm
rooms	Number of rooms in household
nmkat	Net rent category (1: up to 500 DM; 2: 500 DM to 675 DM; 3: 675 DM to 850 DM; 4: 850 DM to 1150 DM; 5: 1150 DM onwards
adr	Address type (1: bad; 2: average; 3: good)
wohn	Residential type (1: bad; 2: average; 3: good)

The subject of interest were the price ranges of apartments with different specifications. The apartments were defined as separate datasets for simplicity, easier accession, and better readability by filtering the original survey by the specified parameters. Subsequently, the range of the net rent in DM variable was called forth, followed by the range of net rent per square meter, rounded to two digits.

The first group of rentals contained one room apartments built after 1966 that were smaller than 50 square meters. The price range of such an apartment at the time of the survey was between 340.00 and 1065.63 DM, with the price range per square meter between 11.16 and 33.89 DM. That is considerably more expensive in comparison to the second group of apartments, consisting of 3 room apartments of the same age category with quadrature of more than 80 square meters. While the price range starts at a comparably low price of 323.03 DM and exceeds the highest value by more than double at 2680.00 DM, the price adjusted to the size of the apartment starts at significantly lower prices of 2.86 DM per square meter and does not exceed the highest price of the first group with 28.93 DM per square meter.

A price range was obtained also for an apartment inspired by a listing from Graz at 523€ or 1022.90 DM. Said flat was chosen to be sized between 50 and 55 square meters, have two rooms and be less than 15 years old at the time of the survey. The prices for such an apartment in 1994 Munich would have been between 351.36 and 1294.30 DM, with the price adjusted to size between 6.51 and 23.97 DM.

The specifications of the apartments and the acquired price ranges can be found in Table 2.

Table 2 Properties and price ranges of specified apartments in the survey "Mietspiegel München(1994) "

Apartment	Floor space in m^2	Rooms	Year of construction	Price range in DM	Price range per m^2 in DM
Apartment 1	Less than 50 m^2	1	After 1996	340.00 – 1065.63	11.16 – 33.89
Apartment 2	More than 80 m^2	3	After 1996	323.03 – 2680.00	2.86 – 28.93
My apartment	50 to 55 m^2	2	After 1980	351.36 – 1294.30	6.51 – 23.97

3.2 Mystery

The mystery file contained three variables – dataset, x, and y, given in a tab separated file. Summary statistics were performed on both the x and y variable, extracting the minimum, maximum, mean, and standard deviation of the value. The summary statistics can be found in Table 3. A correlation between the two variables was established at -0.06601891.

The data was reorganised to account for the numerical sequence and the summary statistics were repeated while grouping the variables by their respective datasets. The results can be found in Table 4. While the mean and the standard deviation of the grouped variables stays the same as when calculated without separation to datasets, the minima and maxima of the variables change drastically. This implies that, while the variables of all datasets are distributed around the same value in comparable variation, the values themselves vary significantly across the datasets.

Table 3 Summary statistics of the x and y variables in the mystery file

Variable	Minimum	Maximum	Mean	Standard deviation
x	15.6	98.3	54.3	16.7
y	0.0151	99.7	47.8	26.8

Table 4 Summary statistics of the x and y variables in the mystery file, grouped by datasets

Dataset	Minimum x	Maximum x	Mean x	Standard deviation x	Minimum y	Maximum y	Mean y	Standard deviation y
d1	22.3	98.2	54.3	16.8	2.95	99.5	47.8	26.9
d2	15.6	91.6	54.3	16.8	0.0151	97.5	47.8	26.9
d3	22.0	98.3	54.3	16.8	10.5	90.5	47.8	26.9
d4	30.4	89.5	54.3	16.8	2.73	99.7	47.8	26.9
d5	31.1	85.4	54.3	16.8	4.58	97.8	47.8	26.9
d6	27.0	86.4	54.3	16.8	14.4	92.2	47.8	26.9
d7	17.9	96.1	54.3	16.8	14.9	87.2	47.8	26.9
d8	25.4	78.0	54.3	16.8	15.8	94.2	47.8	26.9
d9	21.9	85.7	54.3	16.8	16.3	85.6	47.8	26.9
d10	19.3	91.7	54.3	16.8	9.69	85.9	47.8	26.9
d11	20.2	95.3	54.3	16.8	5.65	99.6	47.8	26.9
d12	18.1	95.6	54.3	16.8	0.304	99.6	47.8	26.9
d13	27.4	77.9	54.3	16.8	0.217	99.3	47.8	26.9

The separate datasets were plotted separately on a scatterplot with the variables x and y representing the respective coordinates. As expected, the dots were scattered in various patterns. Summary statistics only give limited data, barely hinting about the distribution of the elements. In all datasets, the mean and the standard deviation of the variables were identical, yet the values were scattered in plenty of different patterns. Plotting the data gives a much more comprehensive result.

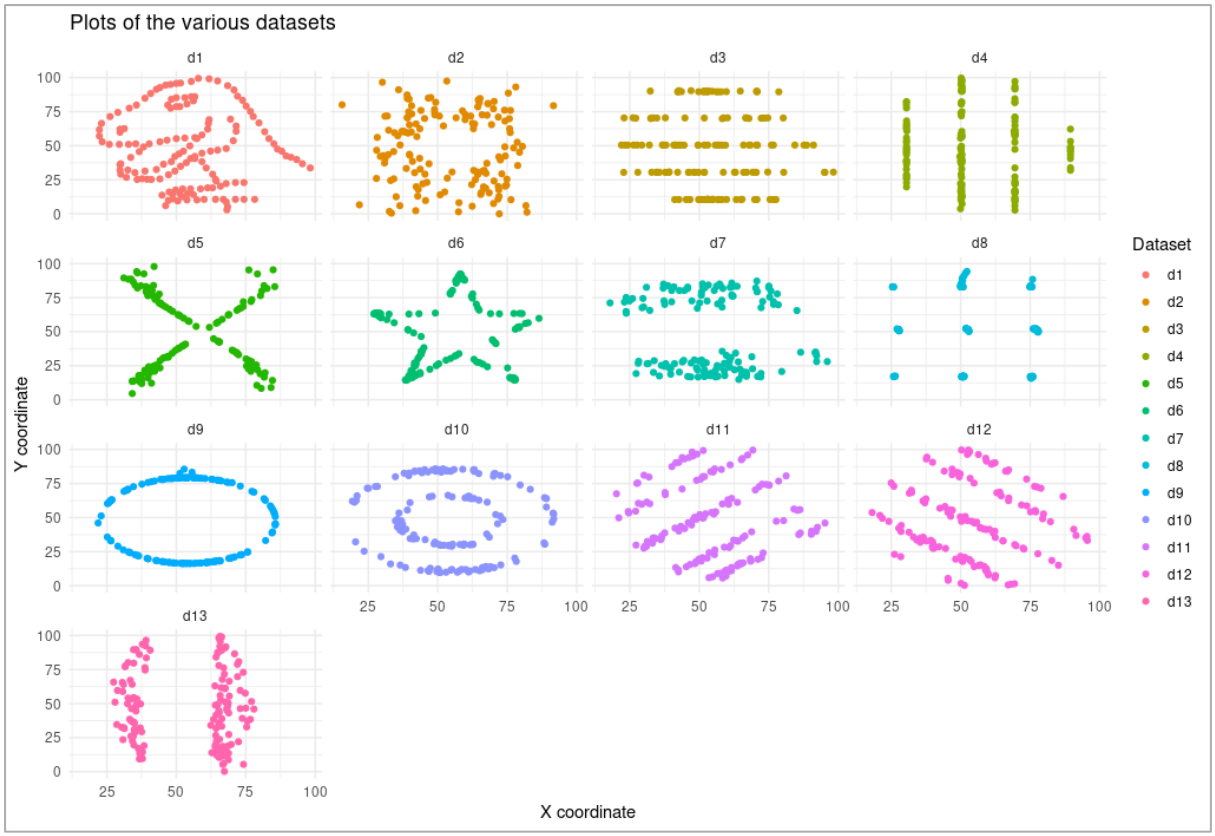

Figure 1 Scatterplots of the x and y variables of the mystery data, grouped by dataset

6 Python scripting 1

6.1 Calculating GC percentages

The percentage of guanine and cytosine in a sequence influences its physical properties, such as melting temperature, and significant discrepancy in the value can indicate e.g. horizontal transfer or different source organism. The python script 6.1_calc_gc_percent.py was written to calculate the percentage of guanine and cytosine in a DNA sequence. The length of the sequence was identified using the `len` command, and the `count` method was used for the count of G and C characters, representing guanine and cytosine, respectively. The percentage of guanine and cytosine in the sequence was calculated by dividing the sum of the count of G and C characters by the length of the sequence. The sequence itself and summary of the results was printed, stating the length of the sequence, the number of G and C bases and the GC content in percent.

6.2 Complementing sequences

In the 6.2_complement.py, a sequence was given, to which a complementary sequence was computed. Thymine (T) was replaced with adenine (A), while guanine (G) was replaced with cytosine (C) and vice versa. The `replace` method was used to replace the uppercase letters representing a nucleic acid of the original sequence with a lowercase letter of their complementary nucleic acid. The returned string was used to then replace another nucleic acid with its complement, repeating the cycle until all nucleic acids of the original sequence were replaced. The lowercase letters were converted to uppercase using the `upper` method, and the original and complementary sequences were printed.

6.3 Extracting exons

With 6.3_extract_exons.py, the exons of the sequence were extracted. Since the start and end of the exon regions were known, exons were extracted from the sequence, while considering that python starts counting at 0. The sequences of the exons were printed individually and together. The exon region percentage was calculated as the sum of the length of the exons, divided by the length of the entire sequence. The original sequence was also printed with introns masked in lowercase letters, by extracting the intron and printing the separate parts of the sequence, utilizing the `lower` method for the intron.

6.4 Splitting sequence and creating FASTA files

A FASTA file was given, containing a transcript of the human gene AMY1A. To extract the three exons from the sequence, the script 6.4_split_seq_to_fastas.py was created. The sequence was opened using an absolute path. The first line of the sequence was ignored using the `readline` method. A blank variable was created, into which the sequence was to be written. Using a for loop, the file was read line by line, stripping it of the line break and adding the line to the sequence string. After processing, since the entire sequence was contained in a string, the file was no longer needed and was closed. To check the sequence extraction was successful, the sequence length was printed. The exons were extracted from the sequence based on their starting and ending base. New files were created using the `open` command, in which the exon regions and intron regions of the sequence were written in using the `write` method. A header was created for each fragment, followed by a line break "`\n`", followed by the fragment sequence in the next line and a line break at the end of the sequence. To check for correct extraction, the length of the fragments was printed. After all fragments were written, the new files containing the exon and intron regions respectively were closed.

6.5 Converting table into FASTA

A table of exons in one of the transcripts of the human BRCA2 gene was given in the file BRCA2-202_exons.tsv, separated by tabs. The table contained the exon ID, chromosome number, start position, end position, length, and sequence of exon. The names of the columns were given in the header, with each row representing a new exon. Using the 6.5_convert_table_to_fasta.py script, the contents of the table were converted into a FASTA format.

The table was opened in an absolute path, and a new file was created for the sequences in a FASTA format. The table was read line by line using a for loop, and the contents of each line were stripped of the line break. A list of the fields for the line was extracted by splitting the line at the tab symbols "`\t`". A variable was created for each field, containing the string of the field. These fields were then used to write the FASTA file. A header was created for each exon, containing the information on the exon ID, chromosome number, start, and end bases and length of fragment, followed by a line break. The sequence of the exon was written in the next line, followed by two line breaks for better readability of the file. After iterating the for loop through each line of the table, both the table file and new FASTA file were closed.

7 Python scripting 2

7.1 Translating DNA into protein, part 1

The 7.1_translate_dna_to_protein.py script was written for translating genetic sequences into proteins. The genetic code was defined as a dictionary variable, assigning each codon the amino acid it is translated to. Stop codons were translated to "_". A function was created for translating genetic sequences to proteins. The last possible starting bas of the codon was defined as the length of the

sequence subtracted by two, as codons are three bases long. The protein sequence was initiated as a blank string. A for loop was started to iterate over possible codon start positions, using the `range` function, starting at 0 as the first position and ending at the previously defined last possible codon start, in steps of 3. In the for loop, codons were assigned to a variable, extracted as a substring from the DNA sequence, from the start position until the end, defined as three characters after start. The amino acid translation was then found with the `get` method in the previously defined dictionary variable. If no match was found, the amino acid was written as X to mean unknown. This translation of the codon was appended to the end of the protein sequence. The return of the defined function was the protein sequence.

A list of DNA sequences was given in a list, and for each sequence in the list, the sequence was printed, followed by the translation by using the above defined function for translating DNA sequences.

7.2 Sorting tabular data

The script 7.2_sorting_tabular_data.py was used to append the length of sequences to the table in the yeast_chip.tsv file and sort the list according to the length. The file was opened, and the header was defined as the first line by `readline`. The element "length" was appended to the first line and the contents were defined as an empty list. A for loop was created to iterate over each line of the file, in which the line was split into fields and the length of the fragment was appended to the line, calculated by subtracting the starting position in the fifth field from the stop position of the fragment in the fourth field, while taking into account the indexing in python starts at 0. The fields in the line were appended to the content list. The new list was established with the elements sorted according to the length in the twelfth element, sorting the content list by the twelfth element with the `sorted` function. A new table was printed, consisting of the first line with the appended length column and the sorted contents list. The yeast_chip.tsv file was closed.

7.3 Calculating word frequencies

7.3_calculating_word_frequencies.py was written to track the frequency of words in a text file. `string` and `Counter` from collections were imported. The text file was opened, read, converted to lowercase, and split into words. Using `maketrans`, a mapping table was made for translating to strip the words of punctuation, and each word was translated using `translate`. The words were sorted by the key of their count in the list in descending order. An empty list was created. The sorted words were iterated over using a for loop, an if loop was used to consider only those with 10 or more occurrences in the list. If the word was not yet in the new list, it was appended to it. The words were printed using a for loop to iterate over the new list, and if the element was also found in the list of sorted words, a message was printed containing the word and the count of the word. Using this script, the most frequent words were discovered: the: 88, to: 39, file: 34, and: 23, a: 20, in: 19, directory: 18, you: 18, of: 18, command: 16, are: 14, file : 14, use: 12, many: 12, how: 12, exonsfasta: 11, you : 10, is: 10, what: 10.

8 Python scripting 3

8.1 Translating DNA into protein, part 2

The script in 7.1 was modified to work on FASTA files and saved as 8.1_translate_dna_to_protein2.py. After defining the dictionary variable and the function to translate DNA sequences into protein, gzip was imported. A function was defined for opening files depending on whether they are gzipped files. An if loop was created to open the file using `gzip.open` if the selected file ended with `".gz"`, else, the file was opened with the `open`

command. To check if the function worked, a statement describing the way the file was opened was printed. The FASTA file was using this function, and another file was opened with writing rights to write the translated sequence in. The translation process was written in a for loop. Iterating over each line in the sequence, the lines were first stripped of the line break. To differentiate between headers and sequence, an if loop was used. The headers were recognised by starting with the ">" character and were printed and written into the new file along with a line break, else, the line was recognised as containing the sequence. This was translated using the previously defined function to translate DNA sequences into protein and written into the new file. Both the sequence and the translated sequence files were closed.

8.2 Counting sequences

The script count_seqs.py was given for counting sequences in the FASTA file NC_000913.faa. It was modified to 8.2_count_seqs.py to do the same for any FASTA file given as an argument. For that, `sys.argv` was used. The script from the original was modified in a while loop to open the file in the argument as the filename. The script was further modified to 8.2_count_seqs_multi.py to do the same for multiple files given in arguments. A for loop was established, which iterated over every argument from the second argument onwards, as the first argument used is the script name.

8.3 Checking for start and stop codons

The script 8.2_count_seqs_multi.py was modified to 8.3.1_check_start_stop_codons.py, instead of counting the sequences in a file given in an argument, rather count the sequences that do not start with M with `record.seq.startswith()`, and printing the ID of detected proteins using `record.seq.id` and the amino acid they start with. Protein sequences usually start with the amino acid methionine (M), because AUG is the start codon, translating to M. If the protein sequence does not start with M, it implies that we do not have the correct or full translation of the gene, or that, in case the sequence is that of a protein post-modification, that it has undergone post-translational modifications.

The script was expanded to 8.3.2_check_start_stop_codons.py to also print the number of sequences in the selected file that do not start with M, by introducing a variable that is increased by 1 if loop, where the sequence does not start with M. It was further expanded to 8.3.3_check_start_stop_codons.py, extracting also the sequence ID, length and features of each sequence not starting with M. Lastly, in 8.3.4_check_start_stop_codons.py, the sequences were searched for those not ending with "*", for which `record.seq.endswith()` was used. A new if loop was started for this, nested on the same level as the one searching for sequences starting with M, and an identical loop was nested under the previous if loop, now searching among the sequences not starting with "M" for those not ending with "*".

8.4 Calculating fragment frequencies

The script 8.4_fragment_frequencies.py was written to search a sequence for all fragments the size of N. First, a function was defined that splits the sequence into N sized chunks. N was defined to be able to easily change the size with only one alteration. An empty list of chunks was defined, as well as the starting position a with the value 0, to start at the beginning of the sequence, and the end position b, the same size as the chunk size. A for loop was established to iterate over every chunk in the sequence, extracting the chunk from the sequence between the positions a and b, while displacing a and b by 1 each iteration. An if loop was created to append only the chunks of size N. The total number of chunks was defined and printed as the final chunk value subtracted by N, to dismiss the smaller chunks from the final for loops, where b was out of range. The function returned a list of chunks of size N.

Next, two functions were defined to search for the most and least frequent chunks. In the function searching for the most frequent fragment, two variables were defined, the counter of 0 and the most frequent chunk, were defined and a for loop was created, iterating over every element in the given list, in which the frequency of the current element was created using the `count` method. An if loop was created, in which the current frequency was compared to the previous counter. If the frequency of the current element was greater, the counter was overwritten with the value of the current count and the most frequent chunk with the current chunk. A message was printed on the most frequent chunk and its count. The function searching for the least frequent chunk operated on the same logic, only comparing the frequencies of the chunks, and searching for the lesser value.

A FASTA file was read using a with statement. An empty string was created, in which genes were transcribed by reading the file line by line, removing the line break at the end of the line. The headers of the genes were skipped using an if not loop, searching only for the lines not starting with ">" and appending them to the string of genes. The file was closed and the sting containing the genes was worked with in further steps.

The function to split the sequence into chunks was performed in the gene string. The returned list of chunks was used with the functions to identify the most and least frequent fragments. Both functions returned a message stating their selected chunks.

9 Regular expressions and python scritping 4

9.1 Selecting matching patterns

A regular expression was given, alongside several patterns. The patterns that matched the regular expression were selected.

1. a(ac)*a – "a" + "ac" in zero or more occurrences + "a"
   Matches: aaca, aa, aacacaca
   Not matching: acaa , acacac
2. CT+G – "C" + "T" in one or more occurrences + "G"
   Matches: CTG, CTTTTTTG
   Not matching: ctg, CG; CTGG
3. a.[gt]+ – "a" + any character + either "g" or "t" in one or more occurrences
   Matches: atg, atttttt, ang, anttggtgg
   Not matching: at
4. acg|TGA – either "acg" or "TGA"
   Matches: TGA, acg
   Not matching: acgGA, acgTGA, acTGA
5. \w{0,10} \d\d?\d? – string with any word characters, either 0 or 10 times + string containing numbers + string containing numbers in zero or one occurrence + string containing numbers in zero or one occurrence
   Matches: 657 019, ILikeNumber 14, lab 04
   Not matching: St Paul 3, Length: 752
6. [a-z]*[\.\?!] – characters from "a" to "z" in zero or more occurrences + one of [\.\?!]
   Matches: wow!, ?
   Not matching: Big, small.?, looking to., When?, actg..
7. >?\w+ [Ll]en=\d* – ">" + string of any word characters on one or more occurrences + " " + "L" or "l" + "en
   Matches: >seq027 len=,
   Not matching: > GSEEEH7 Llen=033, >read_17345, r0274.1239 len=199

8. <[^>]+> – "<" + any character string EXCEPT > once or more + ">"
   Matches: <xml?>, <font family="monospace">
   Not matching: <table></table>, < >
9. ([actgACTG]{4,4})\1+ – ([actgACTG] combination of 4 these characters) repeated once or more
   Matches: AtAtAtAt
   Not matching:TTTTATTTTATTTT, GCCGCCGCC, CATGCATGCATGC, aggtAGGTaggtAGGT, annnannn

9.2 Matching homopolymer tails

The following regular expressions were built in 9.2_matching_homopolymer_tails.py:

- Case-insensitive, matching all poly-A sequences with the length of 6 or more – "[Aa]{6}[Aa]*"
- Case-insensitive, matching all poly-A tails with the length of at least 6 – "[Aa]{6}[Aa]*$"
- All homopolymer tails of at least 6 bases –
  "([Aa]{6}[Aa]*$)|([Gg]{6}[Gg]*$)|([Tt]{6}[Tt]*$)|([Cc]{6}[Cc]*$)"

Five sequences were investigated:
seq1 = "ACTGGGTCTCTAAAGGGTCGTAAAAAACTCTCAAAAAAAAA"
seq2 = "GGTCTCTAAAGGGTCGCCCCCCCCTCTCGGGGGGGGG"
seq3 = "tgggtctctaaagggtcgtaaaaaactccccccctttttt"
seq4 = "CTGGGTCTCTAAAGGGTCGTAAAAAACTCTCCCCCC"
seq5 = "tgggtctctaaagggtcgtaaaaaactccccccccttttaaaaaaa"

Using the regular expressions above, the sequences were investigated on homopolymer sequences. The first sequence was found to contain a poly-A tail, but no other poly-A sequences. Sequence 2 contained a poly-G tail. Sequence 3 contained a poly-A sequence, but not at the end. Sequence 4 contained a poly-A region and a poly-C tail. Sequence 5 contained a poly-A sequence in the middle of the sequence and a poly-A tail.

9.3 Extracting information from matching records

A list of following sequences was given:
PR|00561934|AA:221
PR||AA:450
GN|47671|BP:3134
PR|0017|AA:321
GN||BP:1459
PR|1043169|AA:529
GN|947771|BP:2722

A regular expression matching the sequence in this format is "(PR|GN)[|]\d*[|](AA:|BP:)\d+".The components match the following:

- (PR|GN) – either "PR" or "GN"
- [|] – the symbol |
- \d* - any number in zero or more occurrences
- [|] – the symbol |
- (AA:|BP:) – either "AA:" or "BP:"
- \d+ - any number in one or more occurrences

Since the gene length is always at the end of the sequence, it can be captured using "\d+\Z".