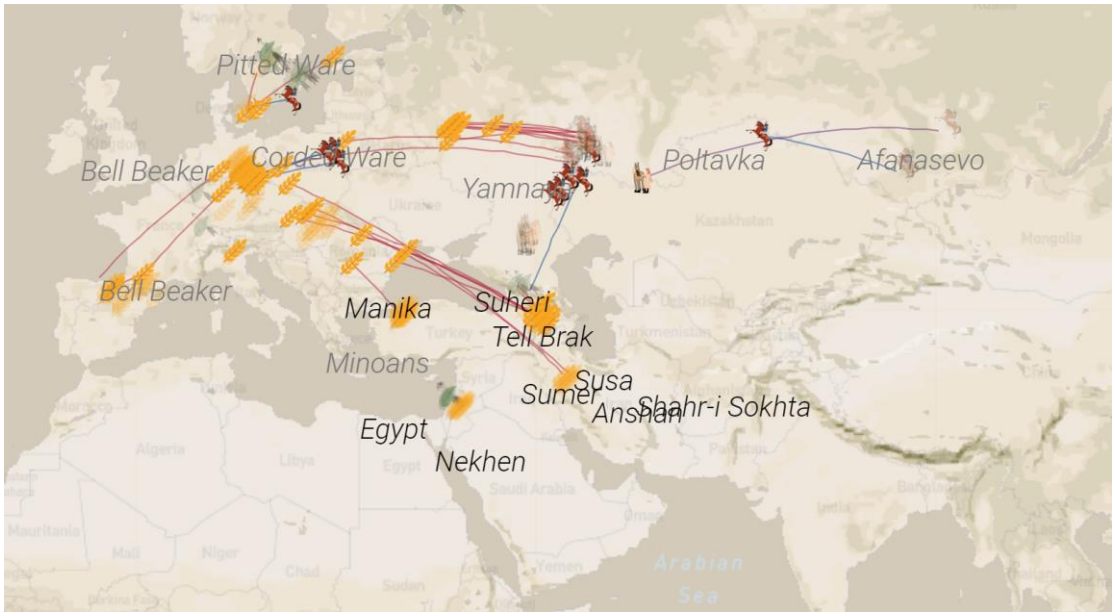
Python Exercise Workbook



From: The Atlas of human migration (2536 BC)

Bioinformatics: Programming in Python (BINP16)

October 2021

Master's program in Bioinformatics

Lund University

Contents

Credits

Edited, revised, and solved the questions: Malou Arvidsson, Arthur Boffelli, Mara Vizitiu, Mattis Knulst, Jiawei Zhao, Julio Lopez, Virag Varga, and Yali Zhang.

1. Simple data type

1.1 Arithmetic

1. Before evaluating the expressions in Python, discuss what these expressions will return:

```
5/0
5//2 (what is the rule?)
5%-2
-5%2
-5%-2 (what is the rule?)
round(-3.6)
round(4.5)
round(5.5) (what is the rule?)
round(6.666,2)
```

   Remark: modulus with negative numbers is treated differently in different programming languages. For regular programmers it is of less (or no) importance.

2. Will these expressions give the same result?

```
round(int(4.6))
int(round(4.6))
```

3. Programming languages often have a function called floor which returns the nearest integer less than the floating point number. There is also a function ceil which does the opposite. The functions are provided by the math module. Don't forget to import the math module. Test:

```
int(-3.1)
math.floor(-3.1) (How to import the module?)
math.floor(3.1)
math.ceil(3.1)
```

4. Conversion of different data types. Try to convert data type between integer, float and string. Test:

```
type(6.66) (What data type it is?)
int(6.66)
float('6.66')
int('6.66') (Could this work?)
```

   Find a way of converting string '6.66' to integer.

1.2 Strings

1. Will this work: len(5)?

2. Figure out what len(str(len('len'))) returns, then test.

3. Exploring string operations.

   a) You start with the variable my_string = "ATGC". What is the most straightforward way of updating the content of my_string to "ATGCXXX"?

   b) Starting with the variable my_string = "ATGC". What is the simplest way of updating it to "ATGCATGCATGCATGC"? (this should be solved in a different way than the previous).

   c) Calculate the length of the updated my_string variable.

   d) Take the string "AAAAGGAAAAGGAAAA". Calculate the position of the first GG.

   e) *Then, find a way of calculating and printing the positions of all GGs in the string. (You might need to check the documentation for this one).

   f) Use the same string as in d). How many occurrences of AAAA are there? AAA? AA? A? Do you understand why?

   g) Store the strings "AcgT" and "acGT" into two different variables, and do a check on whether they contain the same letters. You can do this with a basic comparison after doing a method-call on each of them. This type of problem is common when working with sequence data, where the sequences sometimes contain a mix of upper- and lower-case letters.

4. The format method. This is an extremely useful thing. Learn it! Use it!

   The primitive way to print a combination of strings and variables is to simply concatenate them.

```
name = "Karl"
surname = "Johansson"
print("My name is " + name + " " + surname).
> My name is Karl Johansson
```

   Now, do the same using string formatting. Do the following steps:

   a) Start by making an identical string as the one above, but using string formatting instead.

b) Next, provide position arguments between the curly braces ({0}) to swap position of name and surname in the output.

c) Oh no. We realized that Karl actually prefers to be called KarlKarl. Can you do a very minor change to your existing script to take this into account? Don't change the initial value of name - Karl is still his real name. (You only need to add two characters in one place).

d) Oh no again. Now we realized that KarlKarl prefers to only be called KarlKarl Johan, rather than KarlKarl Johansson. Can you do another very minor change to take this into account? You should still keep the original values of name and surname the same.

5. More string operations. Now you have two nucleoside sequences:

```
seq_1 = 'AAATT ' #There is a space at the end of the nucleotide se
quence

seq_2 = 'CCCGG'
```

Please concatenate these two sequences into one sequence and print 'AAATTCCCGG'. Try different methods.

6. *Ask the user for a nucleotide sequence, mutate a random (real random) nucleotide and print it.

For example:
Input: AAACCC Output: ATACCC

1.3 Numbers

Tips: In this part, you may use loops and conditional statements which you will learn in Lecture 3 program control part. Loops can repeat the block of commands within the loop. Conditional statements can check if some statement is true and continue in several possible ways. Python employs indentation to determine the boundaries of code blocks. Here is the general form and examples for loops ('for', 'while') and conditional statements ('if'):

The 'For' loop:

```
for i in values:
    statement
```

```
for i in range(1,5): # An example
    print(i) # output is: 1 2 3 4
```

The 'while' loop:

```
While expression:
```

```
    Statements
        else:
    statements

i = 1 # An example
while i < 3:
    print(i)
    i=i+1
else
    print('stop') # output is: 1 2 stop
```

<u>The conditional execution 'if':</u>

```
if expression1:
    statement1
elif expression2:
    statement2
else:
    statement3

i =1 # An example
j = 2
if i < j:
    print('i') # output is i
else:
  print('j')
```

1. Print all multiples of 3 below 1000.
2. Print all multiples of 3, except those that are also multiples of 5, between 3000 and 4000.
3. In a Fibonacci series, the new value of the series is the sum of the previous two. It looks like this: 1,1,2,3,5,8,13,21... Ask the user for a number and print that many values from the Fibonacci series.
4. Ask the user for a number and print the factorial. The factorial of 4 (denoted 4!) is 1*2*3*4=24. What is the largest value a user can enter (on your computer)?

   For example:
   Input: 4 Output:24
5. *Print the first 1000 prime numbers.

2. Collection data types: Lists, Sets and Dictionaries

2.1. Lists

1. Ask the user for three ingredients, one after the other, and print a list with the ingredients surrounded by bread: ['bread', 'ingredient1', 'ingredient2', 'ingredient3', 'bread']

2. Use `input()` to get a sequence of numbers (divided by comma). Calculate the sum and print it to standard output.

3. The line below contains all possible codons. Load it as a single string in python and create a list of codons. Make sure they are all uppercase or all lowercase.

```
AAA AAT AAC aag ATA ATT ATC ATG Aca ACT ACC Acg AGA AGT AGC agG TAA TAT TAC TAG
TTA TTT TTC TTG TCA tct TCC tCg TGA TGT TGC TGG CAA CAT CAC CAG CTA cTT ctc CTG
CCA CCT CCc CCG CGA CGT CGC cgg GAA GAT GAC GAG GTa GTT GTC GTG GCA GCT GCC GcG
```

4. From the list created in the previous exercise, remove all codons that result in the amino acid Leucine.

5. Create the same list of codons totally in python starting with the string 'ATCG'. *

> Hint: For loops can also be used to iterate through lists, sets, tuples, and dictionaries.

2.2. Dictionaries

1. This week you have bought four apples, two pears and two oranges. You want to keep track of them using a dictionary. Try two ways of creating this dictionary:

   (1) Define an empty dictionary syntax and then populate it using the assignment syntax.

   (2) Define an already populated dictionary using the curly-braces syntax.

   Print the final results to verify.

a) Now you have bought two more pears, five more oranges and one watermelon. Update the existing values in the dictionary to account for this (note that arithmetic operations work in the same way for dictionary values).

b) Iterate through the keys of the dictionary and for each key print a line showing what fruit it is and how many pieces of that fruit you own. Example output:

```
apple: 25 pieces
orange: 10 pieces
... and so on
```

c) Do the same, but make sure the fruits are printed in alphabetic order.

2. Your friend didn't know about your fruit list, and independently made their own list. They kindly provided you a Python one-liner to load this list into another variable `friend_week_fruits`. Load this into your terminal / script. *

```
friend_week_fruits = {'apples':2, 'pears':1, 'oranges':2,
'waxberry':4}
```

Now, do a joint dictionary summarizing all the fruits. So - all fruits present in either dictionary should be present as keys, and when a fruit is present in both, you want the total sum. Note that this is a bit tricky. You need to consider that each fruit list has unique fruits not present in the other ones. Print similarly to in 1-c) and make sure all counts make sense.

3. Make a tiny tab-separated file containing two columns found below. *

ProteinID	ProteinSeq
prot1	AGSATGDASD
prot4	ASLWASLD
prot9	PPASDSADSAD
prot2	XXSWKJXS
prot8	PSOASSADASD

a) Load the table from file and insert it into a dictionary. Iterate through it in same order of the IDs.

b) Now, iterate through it again, but only print peptides containing Tryptophane (W).

c) Finally, iterate through it, printing entries NOT containing X.

Hint: Files can be loaded in python using the **`open(filename, mode)`** function. The necessary arguments are the path for the file and the mode that the file is opened. The mode can be **'r'** for read, **'w'** for write, and **'a'** for append. An opened file must be closed after used, using **`.close()`** after the variable name of the file. Using **`with open(filename, mode)`** python closes the file automatically.

2.3. Sets

1. Check if all sequence ids are unique in `regions.fna`. *

2. The file `reads_ids.txt` contains sequence ids. Check how many of the ids that occur in the fasta file `reads.fna`. Tip: Test the difference in run time using a list and a set for looking up ids. *

3. Find the number of fasta sequence ids that are present in both regions_sub1.fna and regions_sub2.fna. Then see how many of these ids that are not present in regions_sub3.fna. *

3. Program control and logic

3.1 Loops and conditions

Hint: many of these exercises assume that you assign different values to a variable. This can be hardcoded within the script:

```
#!/usr/bin/python3
my_name = "Jakob"
```

It can also be made interactive - so that the script asks you for the value when running it. (More on this on a coming chapter.)

```
#!/usr/bin/python3
my_name = input("Please enter your name: ")
```

Then, when you run it, it will ask you for input.

```
$ ./my_script.py
Please enter your name: <Enter your name here, and press enter>
```

6. Giving special treatment to yourself. Write a script which first figures out if you or someone else is using it, and then greets the user. If it is you, give yourself an extra nice greeting.

   Use a variable and an if-else statement. Either hard-code the value of the variable, or read it using the input function. Check in an if-statement whether the variable contains yours or someone else's name. Prepare two different print-statements - one for you and one for everyone else.

   When used, it should give an output similar to the following:

   ```
   $ ./greeting.py
   What is your name: Yourname
   So very nice to see you Yourname!
   $ ./greeting.py
   What is your name: Someoneelse
   Hi Someoneelse
   ```

7. Giving different special treatment to your friends. You want to greet your friends too, but realize that you want to keep the very special greeting for yourself.

   Rewrite the script from the previous script so that your friends receive a greeting different from yours. You still want to get the very special greeting yourself, and to give the bland, neutral greeting to people you don't know.

   It should give an output similar to the following:

```
$ ./greeting.py
What is your name: Yourname
So very nice to see you Yourname!
$ ./greeting.py
What is your name: Randomperson
Hi Randomperson
$ ./greeting.py
What is your name: Afriend
What's up Afriend!
```

8. FizzBuzz simplified. This is inspired by an (in)famous coding interview question claimed to filter out the 'top percent who actually can program'. This is probably an exaggeration, but writing a nice solution can be trickier than it first looks!

   Make a script taking a number, and giving one of the following outputs:
   If the number can be divided by three, print "Fizz"
   If the number can be divided by five, print "Buzz"
   If the number can be divided by both three and five, print "FizzBuzz"
   If the number is neither dividable by three nor five, print the number

   The input can either be assigned directly in the script, or read using the input function. In the latter case, the input needs to be converted to an integer:

```
current_value = int(input("Provide the next number: "))
```

   Example usage:

```
$ ./fizzbuzz.py
Provide the next number: 5
buzz
$ ./fizzbuzz.py
Provide the next number: 8
8
$ ./fizzbuzz.py
Provide the next number: 15
fizzbuzz
```

9. Real FizzBuzz. Make your script run over all integers between 1 and 100, and print the correct fizz, buzz, fizzbuzz or value for each of them.

   Example output:

```
1
2
fizz
```

```
4
buzz
...
13
14
fizzbuzz
16
... and so on
```

10. Let's play golf! You have a golf course with 8 holes. In golf, your score depends on the number of strokes required to sink the ball in the hole. Your score is compared to the par and has a different name depending on the difference. For our course, if you used three (or more) strokes less than par, it's called Albatross, two strokes less is Eagle, one less is Birdie, same as par is Par, one more than par is Boogie, and two or more is called a Double boogie. The par scores for your golf course are coded in a list: [4,3,5,2,5,4,7,6]. Make a program that asks the user for the hole they are in (1-8) and their number of strokes. Print out the name of the score in that hole. For example, 3 strokes on hole 4 is a Boogie (one over par).

11. Explore how the enumerate() function works:
    Given the following list of letters, letter_list = ["a", "b", "c", "e", "f", "g", "h", "i"], write a script that will output the uppercase version of every second letter, starting from the first. Use the enumerate() function. The syntax in this case is:

    ```
    for index, letter in enumerate(letter_list):
    ...
    ```

12. *You have received the following mRNA sequence, that consists of 10 codons:

    UAUAAACGAUACCAUUACUAUGACCAUGGG

    You are interested in those codons that encode tyrosine. Knowing that they are "UAU" and "UAC", find out in which positions in the sequence (what is the number of the codon, out of 10) tyrosine is encoded.
    Hint: you can change the numbering of the indexes output by the enumerate() function using the argument start as follows:

    ```
    for index, value in enumerate(my_list, start=1):
    ...
    ```

13. *Print the first 1000 prime numbers. Hint: to test whether a number is prime, it is enough to test whether it is divisible by any prime number smaller than half of its value.

3.2 Error exceptions

14. Write a script that asks the user to provide a DNA sequence and raises an exception if any non-nucleotide letter is found in the sequence.

15. *Modify the previous script so that the user is prompted to re-enter a DNA sequence until all of the nucleotides are valid. Display a message once a valid sequence has been entered successfully.
    Example of output:

    Please type a DNA sequence: ACCAB
    Your sequence contains an invalid nucleotide. Please try again.

    Please type a DNA sequence: AGGTAHT
    Your sequence contains an invalid nucleotide. Please try again.

    Please type a DNA sequence: GATC
    You have entered a valid DNA sequence.

4. File handling

To solve these problems you need to be familiar with open(), file formats, shebangs, sys.argv and pythonic string manipulation methods.

Don't worry if you need to work on these things, search for the above terms (e.g. python open()) or ask an instructor to help explain them to you. Most IDEs will display documentation for functions as you type them, remember to use documentation to get an idea of what you can do without reinventing the wheel!

1. Python natively supports using file addresses with forward slashes on any system, but there are cases where you need to do things to those paths where string manipulation just becomes a hassle.

   This is especially true if you want to make your scripts work with paths in different operating systems. Windows infamously uses backslashes for its own paths and if you write a script for a friend or colleague, they may just want to paste a path into the command line.

   You may also be interested in listing files, directories or going up and down file trees. The module for this used to be os, but there is a newer module in the standard library that is extremely powerful for working with paths on any OS: pathlib.Path

   a) import Path from pathlib

      hint: almost the same as the above line, but Python syntax demands different order!

      b) Store your current working directory and home directory in variables and print them, also print their type().

      hint: start by typing Path and scroll through the different methods or look in the docs for pathlib!

      c) Make a new directory by joining the string "new_dir" with the parent of your working directory

      hint: you need Path.parent, Path.joinpath() and Path.mkdir()

      d) List all directories in your home directory that contain a letter that you choose! Print these to screen!

      hint: use Path.glob() or Path.iterdir()

e) Use the same method as in d) but this time only extract file paths matching a pattern and print only the file name, excluding the path and file extension, along with its type!

hint: Path.stem and Path.is_file() will be useful here!

f) Make your own Path object from a string, the path may or may not exist, use pathlib to check if it does and print a nice message that lets the user know!

hint: use Path() and Path.exists()

2. As a bioinformatician you are guaranteed to encounter many file formats. Generally, the files you encounter will be of one of two types. They can be text files, or binary. Text files are easy to read and parse in Python if they use an encoding that Python can recognize. Binary files can generally only be read with libraries that unpack the information and make it accessible in your programming language. Fasta/q files are required by GenBank to be ASCII encoded.

   a) **Save the string "är det du som har kaffebrödet, Gösta?" to a variable, then use the string method str.encode() to make a new variable that is ASCII encoded. Print both.**

   hint: if you are having trouble copying text from this document, go to the example python scripts!

   b) In the above example you should get either an error message or, if you ignore errors, you'll see a slightly different string than you may have expected. These are the kind of shenanigans encoding can cause. Beyond encoding, text files also tend to have a format, which is documented. In a scientific setting you will see a lot of CSV tables. The CSV format is a table where every column is separated by a comma, but sometimes by other separators such as semicolon. **Create a simple CSV table and write it to a file. Have at least one column with integer or float values.**

   hint: don't make a huge table, 4x4 should be enough, you can use the dictionary from the example script if you think it is hard to come up with one!

   c) Reading and cleaning up CSV files is a huge topic but will be elaborated on in later courses. For the future remember that you will probably want

to install Pandas when you are working with CSV files. Instructions for Pandas are in the extended exercise document. Your school computer should have conda installed, otherwise follow the suitable instructions at https://conda.io/projects/conda/en/latest/user-guide/getting-started.html once you have set up conda you can use anaconda-navigator or the CLI tool: conda install pandas
hint: once pandas is in your environment you can import it in your script, pandas.read_csv() will give you a dataframe object that has methods for all these values! The dataframe is structured like a dictionary where column names can be used to access the columns values.

Read your csv file, then find the mean, median, max and min values for your numerical column and print them!

Use the paxillus.fna file for exercise 3 and 4!

3. Make a script reading the paxillus.fna file, extracting IDs (exclude the > sign and trailing information after whitespace in headers) and printing it to another file. Make it possible to provide the input and output files as argument so that your script can be run as follows:

   hint: str.startswith(), str.strip(), str[:] slicing, and sys.argv are useful!

   ```
   ./parse_ids_from_fasta.py  input.fa  output_ids.txt
   ```

   Do the same script both with and without using the with statement. Which one do you prefer? Why?

4. Let's parse paxillus.fna again. Do a script that reads a FASTA file and then calculates GC-content for each sequence before writing a new file with that information appended.

   hint: string objects have count methods!

   Example input:

   ```
   >id_for_my_entry  additional_info
   GGCCAA
   ```

   Example output:

   ```
   > id_for_my_entry  additional_info  GC:66.7%
   GGCCAA
   ```

   This script should be runnable in the same fashion as the previous. (This means using sys.argv or argparse!)

5. Write a program that takes a fasta file as first argument and a sequence ID as second argument. Search for the sequence ID in the file and print the line number of the sequence. If the ID is not contained in the file, print an informative error message.

   hint: download a few fasta files from databases that you are familiar with, look at headers and try to come up with a solution that is as general as possible!

6. Write a program that converts a fastq file to a fasta file.

   hint: make sure you are very familiar with the file format specifications, for parsing fastq, modulo operations can be useful!

5. Functions

1. Make a function containing a single line of code printing "This is a function!". Example:

```
> print_line()
This is a function!
```

2. Take the function from (1) and make it return a string with the content “This is a function!” instead of directly printing it.

```
> result = print_line()
> print(result)
This is a function!
> print(result[0])
T
```

3. Make a function taking two names and returning a string containing a nice greeting.

```
> print(greet("Björn", "Dag"))
Hello Björn and Dag!
```

4. Change it so that if no people are added it prints question marks, but it still can be used just the same.

```
> print(greet())
Hello ? and ?!
> print(greet("Petr"))
Hello Petr and ?!
> print(greet("Björn", "Dag"))
Hello Björn and Dag!
```

5. Make two separate functions for addition and multiplication of two numbers.

```
> print(multiply(2,3))
6
> print(add(2,3))
5
```

6. Now merge them into a single function which takes two numbers and a third argument specifying what type of operation to perform.

```
> print(calculate(2, 3, operation="add"))
5
> print(calculate(2, 3, operation="multiply"))
6
```

7. Make a function able to take a string of nucleotide letters and return its GC content.

```
> print(get_gc("CACAGGTT"))
0.5
> print(get_gc("CAG"))
0.6666…
```

8. Make an obnoxious function printing "Hi!" a specified number of times.

```
> many_hi(4)
Hi!
Hi!
Hi!
Hi!
```

9. Now change it so that it returns a list with the specified number of "Hi!".

```
> hi_list = many_hi(6)
> print(hi_list)
["Hi!", "Hi!", "Hi!", "Hi!", "Hi!", "Hi!"]
```

10. Finally, add an optional parameter so that you can change "Hi!" to another phrase of your choosing.

```
> hi_list = many_hi(3)
> print(hi_list)
["Hi!", "Hi!", "Hi!"]
> word_list = many_hi(4, word="Halloa")
> print(word_list)
["Halloa!", "Halloa!", "Halloa!", "Halloa!"]
```

11. Create a dictionary containing information from a FASTA-file, with IDs as keys and sequences as values. The input FASTA should look something like the following (feel free to copy it). No function is required - simply store the data in a dictionary.

```
> header1.1
ATGCTAGCTAGCTAGCTACG
> header1.2
ACGTAGCTAGCTAGCAC
> header2.1
AGCTAGCTAGCTATTATCTACT
```

12. Now iterate through the dictionary containing the FASTA and for each step print a message like shown below. It might be useful to iterate the keys and use these to extract the value in each step.

```
Entry 1 has header: header1.1 and sequence: ATGCTAGCTAGCTAGCT
ACG
Entry 2 has header: header1.2 and sequence: ACGTAGCTAGCTAGCAC
Entry 3 has header: header2.1 and sequence: AGCTAGCTAGCTATTAT
CTACT
```

13. Now write a function taking a file path, and returning a dictionary containing a FASTA file such as the one above. You can assume the input is in single-line format.

```
> fasta_dict = load_fasta("sequences.fasta")
> print(fasta_dict)
{'header1.1': 'ATGCTAGCTAGCTAGCTACG', 'header2.1':
'AGCTAGCTAGCTATTATCTACT', 'header1.2': 'ACGTAGCTAGCTAGCAC'}
```

14. Calculate the GC-content for each sequence in the file sequences.fasta. First save all sequences to a list. Then iterate over the list of sequences and call a function that calculates the GC-content. Example code:

```
def calcGC(seq):
    ...
    ...
    return gc

for sequence in sequenceList:
    gc = calcGC(sequence)
    print('{}: {}'.format(sequence, gc)
```

6. Regular expressions

1. Write a program that take an arbitrary string as input and prints contains lowercase letters, contains uppercase letters, contains numbers, contains whitespace if the input string contains any of these character classes, or contains none of these if it contains none of the classes. If the string contains multiple of the character classes above, all relevant contains… strings should be printed!

Hint: apply re.search(pattern, string) function to input.

2. This time someone sent us a FASTA file (regions.fna that we used in Chapter 6) with many records, but we are only interested in records containing a certain sequence. Write a Python program that takes a FASTA file and a sequence as input, and writes records containing the sequence to a new FASTA file. It should be possible to run the script as follows:

```
./extract_fasta_records.py regions.fna ATCTCTC \
    interesting_sequences.fna
```

3. In practice a bioinformatician often gets a file from a collaborator and needs to extract the information she needs from it. In this case we have the file pgi.tre (available on L@L) and want to extract the species identifiers from it (i.e., B2CBB8_Poa_nemoralis_Poaceae). Write a Python program that extracts all species identifiers from the file using a regular expression, and prints them to a new file, one identifier per line. It should be possible to run the script as follows:

```
./extract_identifiers.py pgi.tre identifiers.txt
```

How many identifiers are in the input file?

Hint: Because the input file consists of several exceptions, you should consider all situations into account to avoid unexpected errors.