DA2004/DA2005: Labs

Lars Arvestad, Evan Cavallo, Christian Helanow, Anders Mörtberg, Kristoffer Sahlin

Contents

Lab rules

- All deadlines are **strict**. If you miss a deadline, you must redo the lab or project in the next interation of the course. Contact the course leader if this happens.
- If you know that you will not be able to finish a lab or project (for a legitimate reason such as sickness), inform the course leader **before** the deadline. If you only contact the course leader after

the deadline, your lab will be counted as missed and you will have to hand it in during the next iteration of the course.

- You must work individually on the labs and the project. This means that you should write your own code and find your own solutions. You should not give solutions to each other or copy code from the Internet. All submissions are compared automatically and suspected cheating will be reported to the university's disiciplinary board, which can lead to suspension.
- You can search the Internet for Python commands, syntax, error messages, etc, but not for ready-made solutions. Search in English to get the most answers. You can find answers to many programming questions on the website https://stackoverflow.com/.
- Do not use functions from any library in a lab (that is, do not use `import` anywhere) unless you are given explicit permission.
- If you have a problem with anything other than programming itself, for example if you have a problem using the terminal, conflicts between libraries, etc, contact one of the teachers for help.
- Yor solutions should have a sensible structure. For example, you should not submit extremely long programs with the same copied code over and over instead of a loop. Unnecessarily complicated code can lead to loss of points.

Points

- Every lab is worth 10 points.
- You need to earn **at least 2 points per lab and a total of at least 25 points** to receive credit for the lab course.
- If you earn at least 35 points you will receive 1 bonus point on the project, and if you earn at least 45 points you will receive 2 bonus points.

Submission

- Submissions must be written in Python, not any other programming language.
- Solutions should be submitted in the form of `.py`-files. No file format other than `.py` will be accepted unless specified explicitly in the instructions.
- Solutions must be written in Python 3. It is not permitted to submit Python 2 code.
- Your solution should give the correct output for a given input.

Peer grading on PeerGrade

- Peer grading is a mandatory part of each lab. If you do not peer grade, your lab will be counted as incomplete and you will need to redo the lab during the next iteration of the course.
- Be polite and give constructive feedback on others' code when you peer-grade. If you have a problem, contact the course leader.

Oral presentation

- In most cases no presentation of your submission is necessary, but if something is unclear we can request an oral presentation.
- In a oral presentation, you should be able to answer questions about your solution.

Chapter 1

Temperature conversion

The main idea for this lab to get you familiar with the development environment either in the classrooms or on your own computer.

To receive credit for the lab, you must earn at least 2 of 10 points, but remember that you need to earn at least 25 of the total 50 points across all labs by the end of the course to receive credit for the lab part of the course.

1.1 Learning goals

You should be able to write, run, and make changes in a small Python program.

1.2 Task

The program `convert.py` in the folder "Code for labs" on the course site is designed to ask for a temperature in Fahrenheit, read it from the user, convert it to the Celsius scale, and then print out the result. Here are your tasks:

1. Download, test-run, and study the `convert.py` program from the course site (see the file "Code for labs"). Does it work as intended?
2. Rewrite the function `fahrenheit_to_celsius` so that it calculates correctly. (2 points)
3. Extend the program with a function that converts from Celsius to Fahrenheit. (2 points)
4. Extend the program so that it asks which conversion you want to do. (2 points)
5. Extend the program so that it continues to ask for conversions until the user exits by entering `q` (short for `quit`). (2 points)
6. Extend the program so that it can also convert to and from Kelvin, both from and to Celsius and Fahrenheit. For full credit, the program should continue asking which conversion to do until the user inputs `q`. (2 points)
7. Submit your solution on PeerGrade.io. If you have not registered yet, do so using the signup code found on the course site. **Important:** use the *same name* that you are registered for the course with when you register on PeerGrade!
8. Peer review others' solutions on PeerGrade.io! This will become possible after the submission deadline.

Remember: it is always good to comment your code where necessary (to clarify the purpose of a line of code or a code block) and to document functions. You should also test all the functions you write carefully so that you know they run as you expect!

Chapter 2

Polynomials

In this lab, we will represent polynomials using lists of coefficients. The word "representation" here means approximately "way of storing in a computer". Concretely, we will represent a polynomial like `1+3x+7x^2` in Python as the list `[1,3,7]`. In general, we will store the coefficient of degree `n` in the list's `n`th entry.

2.1 Learning goals

- You will see how to give an abstract concept (like a polynomial) a concrete representation in a computer and how to do calculations with those representations.
- You should be able to write small, simple functions.
- You should be able to work with the data structure `list`.

2.2 Submission

The lab should be submitted as usual via PeerGrade. Don't forget to peer-grade after the submission deadline!

Remember: it's always a good idea to add a comment where it would help explain the purpose of a line or code block to someone reading your code. Likewise, document your functions!

To receive credit for the lab, you must have at least 2 of 10 points, but remember that you must have at least 25 of the maximum 50 points on the labs at the end of the course to get credit for LABO.

2.3 Tasks

In this lab, we will represent polynomials as lists. Below are some more examples:

Polynomial	Python representation
x^4	`[0, 0, 0, 0, 1]`
$4x^2 + 5x^3$	`[0, 0, 4, 5]`
$5 + 4x + 3x^2 + 2x^3 + x^4$	`[5,4,3,2,1]`

Obs: you can assume that the lists you work with only contain numbers.

The Python file `polynom.py` (see "Code for labs" on the course site) contains a function, `poly_to_string`, which converts a polynomial represented by a list to a string.

To get started with the lab, create a file `labb2.py` and copy over the function `poly_to_string`. Complete the tasks below by adding the necessary code to solve the specified problems. As you will see, you will also need to modify the function `poly_to_string` (see task 2).

Obs: you may not use functions from any library in the lab, i.e. you should not use the "import" keyword anywhere in your solution.

2.3.1 Task 1 (0 points, but necessary for tests later in lab)

Suppose that the polynomials `p` and `q` are defined as below.

$$\begin{aligned} p &:= 2 + x^2 \\ q &:= -2 + x + x^4 \end{aligned}$$

Write Python code to store list representations for these two polynomials in the variables `p` and `q`. That is, write

```
p = [...]
q = [...]
```

with the contents of the lists filled in. Check that you encoded the polynomials correctly using `poly_to_string`. You should get the following:

```
>>> poly_to_string(p)
'2 + 0x + 1x^2'

>>> poly_to_string(q)
'-2 + 1x + 0x^2 + 0x^3 + 1x^4'
```

Here, we write `>>>` for the "prompt" of the Python interpreter: `poly_to_string(p)` is a command that should be run by Python, and the lines after are the results. The prompt may look different depending on your computer and interpreter. You can also check the results by adding

```
print(poly_to_string(p))
```

to your file and running it, then observing what is printed in the console. In this case you should not see `'2 + 0x + 1x^2'` but rather `2 + 0x + 1x^2` (without surrounding quote marks). This type of test is very useful when developing code, but you should make sure to remove any `print` calls used for testing in the final code you hand in. It can however be good to add comments with tests and expected results to make it easy for someone reading your code to go through and test it.

2.3.2 Task 2 (3 points)

Edit the `poly_to_string` function so that:

- The empty list is converted to `0`.
- Terms with coefficent `1` are written without a coefficient. For example, `1x^2` should instead be written `x^2`.

- Terms with coefficient `-1` add a minus before the term, but the `1` is not written. For example `2 + -1x^2` should instead be `2 + -x^2`.
- Terms with coefficient `0` are not written. For example, `0 + 0x + 2x^2` should be simplified to `2x^2`.
- A list that contains only `0`s as elements, for example `[0, 0, 0]`, should be written as `0`.

Test your function! Here are some examples with the expected output:

```
>>> poly_to_string(p)
'2 + x^2'

>>> poly_to_string(q)
'-2 + x + x^4'

>>> poly_to_string([])
'0'

>>> poly_to_string([0,0,0])
'0'

>>> poly_to_string([1,2,3])
'1 + 2x + 3x^2'

>>> poly_to_string([-1, 2, -3])
'-1 + 2x + -3x^2'

>>> poly_to_string([1,1,-1])
'1 + x + -x^2'
```

2.3.3 Task 3 (2 points)

a) Write a function `drop_zeroes` that removes all zeros at the end of a polynomial and **returns** the result. *Tips*: use a `while`-loop and the `pop()` function.

```
def drop_zeroes(p_list):
    # here be code
```

Define some polynomials with zeroes at the end, for example:

```
p0 = [2,0,1,0]      # 2 + x^2 + 0x^3
q0 = [0,0,0]        # 0 + 0x + 0x2
```

and use these to test your function:

```
>>> drop_zeroes(p0)
[2, 0, 1]
>>> drop_zeroes(q0)
[]
>>> drop_zeroes([])
[]
```

b) Write a function that tests when two polynomials are equal by ignoring all zeroes at the end and then comparing the rest for equality:

```
def eq_poly(p_list,q_list):
   # here be code
```

Example tests:

```
>>> eq_poly(p,p0)
True
>>> eq_poly(q,p0)
False
>>> eq_poly(q0,[])
True
```

Obs: The functions `drop_zeroes` and `eq_poly` should **return** their results, not just print them out. The difference can be difficult to understand for a beginner, since the result may look the same when you run the code, but there is a big difference between a function that returns something and one that just prints something. See the end of section 2.5.1 in the compendium for more on this.

Note that the code you were given for `poly_to_string` returned the result as a string. Does your solution to Task 2 look similar? If not, go back and make sure you are returning.

2.3.4 Task 4 (2 points)

Write a function named `eval_poly` that takes a polynomial and a value in a variable `x` and **returns** the polynomials value at the point `x`.

Suggestions for your algorithm:[1]

- Iterate over the polynomial's terms by iterating over the coefficients.
- Keep track of the degree of the current term and the sum of the values of the terms you have seen so far. In each step of the iteration, calculate the value of the term as `coeff * x ** grad` (recall that `**` is exponentiation). Then add the term's value to the sum.
- When you have finished interating, return the final sum.

Sample tests:

```
>>> eval_poly(p,0)
2
>>> eval_poly(p,1)
3
>>> eval_poly(p,2)
6
>>> eval_poly(q,2)
16
>>> eval_poly(q,-2)
12
```

2.3.5 Task 5 (3 points)

a) Define negation of polynomials (that is, flip the sign of all coefficients and **return** the result).

[1] The following algorithm is not the most efficient one. If you want to optimize, you can instead implement Horner's method: https://en.wikipedia.org/wiki/Horner%27s_method

```
def neg_poly(p_list):
    # here be code
```

b) Define addition of polynomials (that is, add the coefficients and **return** the result).

```
def add_poly(p_list,q_list):
    # here be code
```

c) Define subtraction of polynomials.

```
def sub_poly(p_list,q_list):
    # here be code
```

Tips: notice that `p - q` can be defined as `p + (- q)`. That is, to subtract the polynomial `q` from `p` you can first take the negation of `q` and then add it to `p`.

Sample tests:

```
# p + q = q + p
>>> eq_poly(add_poly(p,q),add_poly(q,p))
True

# p - p = 0
>>> eq_poly(sub_poly(p,p),[])
True

# p - (- q) = p + q
>>> eq_poly(sub_poly(p,neg_poly(q)),add_poly(p,q))
True

# p + p != 0
>>> eq_poly(add_poly(p,p),[])
False

# p - q = 4-x+x^2-x^4
>>> eq_poly(sub_poly(p,q),[4, -1, 1, 0, -1])
True

# (p + q)(12) = p(12) + q(12)
>>> eval_poly(add_poly(p,q),12) == eval_poly(p,12) + eval_poly(q,12)
True
```

Obs: The comments are only there to explain what these tests are testing. Can you think of more good tests that might turn up in your code?

2.3.6 Task 6 (0 points)

Read through, clean up, and document your code. In order that grading be objective, you should not include your name in the file you hand in.

Tips: read through "Basic principles of programming" under Resources on the course site for recommendations on how to write good code.

Chapter 3

Iteration, file handling, error handling, and dictionaries

This lab contains several independent tasks involving important basic algorithms, dictionaries, file handling, and error handling.

To receive credit for the lab, you must earn at least 2 of 10 points, but remember that you must have at least 25 of the maximum 50 points on the labs at the end of the course to get credit for LABO.

3.1 Learning goals

- You should be able to translate a description of an algorithm to code.
- You should be able to work with dictionaries.
- You should be able to read and write data from files.
- You should be able to use error handling.

3.2 Submission

The lab should be submitted as usual via PeerGrade. Don't forget to peer-grade after the submission deadline!

Remember: it's always a good idea to add a comment where it would help explain the purpose of a line or code block to someone reading your code. Likewise, document your functions so that a reader can easily understand what inputs they take and what the function does.

You may not use functions from any library in the lab, i.e. you should not use the "import" keyword anywhere in your solution. Submit all your code in a single file as in previous labs: functions first, then the main program that calls the functions. In Task 1, you may not use built-in functions for sorting, such as `sort` or `sorted`.

Remember to remove your name from your code if for example Spyder adds it to your file.

3.3 Tasks

3.3.1 Task 1: Insertion sort (2 points)

Insertion sort is a common sorting algorithm, that is, a method for sorting the elements of a list. The idea in this algorithm is similar to one way of sorting a deck of cards: take cardsq from the original deck one at a time and add each card to a new sorted deck of cards in the correct position.

We can divide up the insertion sort algorithm into two subproblems:

a) Write a function that inserts an element `x` in an already *sorted* list `sorted_list` in an appropriate position so that the list remains sorted:

```
def insert_in_sorted(x,sorted_list):
    # here be code
```

Algorithm idea

1. Assume that `sorted_list` is sorted.
2. Iterate over all indices `i < len(sorted_list)` until you find an element `sorted_list[i]` such that `sorted_list[i] > x` and insert `x` before that element.
3. If there is no `sorted_list[i]` that is larger than `x`, then insert `x` at the end of the list.

Sample tests:

```
>>> insert_in_sorted(2,[])
[2]
>>> insert_in_sorted(5,[0,1,3,4])
[0, 1, 3, 4, 5]
>>> insert_in_sorted(2,[0,1,2,3,4])
[0, 1, 2, 2, 3, 4]
>>> insert_in_sorted(2,[2,2])
[2, 2, 2]
```

b) Write insertion sort with the help of `insert_in_sorted`:

```
def insertion_sort(my_list):
    # here be code
```

Algorithm idea

1. Initialize a variable `out` with an empty list.
2. For each element `x` in `my_list`, insert it in `out` with the help of `insert_in_sorted`.
3. Return `out`.

Sample tests:

```
>>> insertion_sort([12,4,3,-1])
[-1, 3, 4, 12]
>>> insertion_sort([])
[]
```

Obs: to recieve credit, you must use `insert_in_sorted` in `insertion_sort`.

3.3.2 Task 2: sparse matrices (1 point)

We can represent a matrix in Python as a list whose elements are lists of numbers all of the same length. For example, the matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 5 \end{bmatrix}$$

can be represented as `[[1, 0, 0, 2], [0, 8, 0, 0], [0, 0, 0, 5]]`.

A matrix is called *sparse* if most of its entries are 0. If we represent such a matrix as a list of lists, we could unnecessarily use up a lot of the computer's memory, especially if the matrix is very big—imagine a matrix with many millions of rows and columns that only contains a handful of elements that are not 0. A better way of representing matrices like this is as a dictionary from coordinates to non-zero elements.

If the coordinates are of the form (row, column) and we begin counting from zero, then the matrix above can be written in the following way as a dictionary:

```
{(0, 0): 1, (0, 3): 2, (1, 1): 8, (2, 3): 5}
```

Write a function `matrix_to_sparse` that takes in a matrix represented as a list of lists and produces a dictionary as above. You can assume that the matrix is well-formed (that is, all the lists inside it have the same length).

Sample tests:

```
>>> matrix_to_sparse([[1,0,0,2],[0,8,0,0],[0,0,0,5]])
{(0, 0): 1, (0, 3): 2, (1, 1): 8, (2, 3): 5}
>>> matrix_to_sparse([[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]])
{}
>>> matrix_to_sparse([[0,0],[0,0],[0,0],[0,10]])
{(3, 1): 10}
```

3.3.3 Task 3: file handling (1 point)

Write a function `annotate(f)` that takes a filename `f` as a parameter and writes to a new file `annotated_f` so that each line in `annotated_f` contains the original line in `f` followed by the row number (counting from `0`) and total number of words up to and including that row.

Example: if the file `infile.txt` contains:

```
A Dead Statesman

I could not dig; I dared not rob:
Therefore I lied to please the mob.
Now all my lies are proved untrue
And I must face the men I slew.
What tale shall serve me here among
Mine angry and defrauded young?
```

then running `annotate('infile.txt')` should produce a file `annotated_infile.txt` that contains:

```
A Dead Statesman 0 3
```

```
 1 3
I could not dig; I dared not rob: 2 11
Therefore I lied to please the mob. 3 18
Now all my lies are proved untrue 4 25
And I must face the men I slew. 5 33
What tale shall serve me here among 6 40
Mine angry and defrauded young? 7 45
```

3.3.4 Task 4: searching for strings in files (2 points)

a) Write a function `find_matching_lines(h,s)` that takes a file **handle** `h` and a string `s`. The function should return, in the form of a list of tuples, both the row number (counting from `0`) and the contents of each row that contains the string.

Example (with `infile.txt` as above):

```
>>> hinfile = open('infile.txt')
>>> find_matching_lines(hinfile, 'the mob')
[(3, 'Therefore I lied to please the mob.\n')]
>>> hinfile.close()
>>> hinfile = open('infile.txt')
>>> find_matching_lines(hinfile, 'the')
[(3, 'Therefore I lied to please the mob.\n'), (5, 'And I must face the men I slew.\n')]
>>> hinfile.close()
>>> with open('infile.txt') as h:
...     print(find_matching_lines(h, 'sommar'))
[]
```

Obs: your search should be *case sensitive*, that is, "the" should not be considered the same as "The". **Tip:** How does `in` work for strings?

Obs: `find_matching_lines` should take a file handle, so it should not contain any call to `open` but rather should assume that `open` was called before the function itself was called, as in the tests above.

b) Write a function `find_lines()` that *asks* the user for a file and string and uses the function `find_matching_lines` to print out the rows where the string was found.

Example: an execution of `find_lines()`, with `infile.txt` as above, could appear as follows:

```
Hello, which file do you want to search in? infile.txt
Ok, searching in "infile.txt".

What do you want to search for? the
The result after searching for "the" is:

Line 3: Therefore I lied to please the mob.
Line 5: And I must face the men I slew.
```

If is up to you whether the program continues to ask repeatedly for input and whether the user can change the file to search in, etc. As long as the user can choose the file and string and you use `find_matching_lines` in your code, you will get credit.

3.3.5 Task 5: searching by position in files (4 points)

a) Write a function `save_rows(h)` that takes a file handle `h` and saves the line numbers as keys and lines themselves as values in a dictionary. The function should return the dictionary.

Example: if the input file `infile2.txt` contains:

```
Hey you
the moon revolves around earth
two chairs and the table
```

then the following should return a dictionary as shown:

```
>>> with open('infile2.txt') as hinfile:
...     print(save_rows(hinfile))
{0: 'Hey you', 1: 'the moon revolves around earth', 2: 'two chairs and the table'}
```

Obs: note that there are no `\n`s at the end of the strings in the output.

b) Write a function `lookup` that takes in a dictionary `d` from numbers to strings as above as well as two coordinates `(r,c)` which correspond to a row and column in `d` and returns the character at the position corresponding to the coordinates.

The idea is that `lookup` will be used together with `save_rows` (see part **c)**) and each character in the file can be said to be at a certain row and column. For example, the word "`Hey`" in `infile2.txt` occupies the three coordinates `(0,0)`, `(0,1)`, `(0,2)`. In the same way, the word "chairs" takes up the coordinates `(2,4)`, `(2,5)`, ..., `(2,9)`.

We assume a 0-indexed coordinate system (as is usual in programming).

The following cases should be handled specially:

- If the row and/or column does not exist in `d`, then the program should raise a `LookupError`.[1]
- If the position is a space, then the string "Space" should be returned.

Sample tests:

```
>>> with open('infile2.txt') as hinfile2:
...     d = save_rows(hinfile2)
...     print(lookup(d,0,0))
...     print(lookup(d,2,9))
...     print(lookup(d,2,10))
H
s
Space
```

But if one calls for example `lookup(d,3,0)` or `lookup(d,0,7)`, then the function should raise a `LookupError`.

c) Use `save_rows` and `lookup` to write a code snippet (a "program") that

1. Asks the user for a file and reads in the file to a dictionary (using `save_rows`)
2. Asks the user to give coordinates for a row number and column number.
3. Uses `lookup` to fetch and then write out the character in the file at that coordinate.

[1] For documentation of the built-in exceptions in Python, see: https://docs.python.org/3/library/exceptions.html

The user should be able to give several coordinates until they choose to write `exit`, whereupon the program should exit. You can decide how to structure your code snippet yourself; for example, you can define a function `main()` that takes no parameters but contains code as described in 1-3 above:

```
def save_rows(...):
   ...

def lookup(...):
   ...

def main():
   infile2 = input('Ange en fil: ')
   indexed_file = save_rows(...)
   # More code here for steps 2 and 3
```

If `lookup` raises an exception because the coordinates are outside the text, then it should be caught with `try-except`. Then the program should print the message `Warning: Out of bounds, try again!` and continue to execute.

Example: if the user runs the program with the file `infile2.txt` as above, then an execution of the program should look like so:

```
At any point type "exit" to quit.

Provide row: 0
Provide column: 3

Space

Provide row: 1
Provide column: 1

h

Provide row: 3
Provide column: 1

Warning: Out of bounds, try again!

Provide row: 2
Provide column: 9

s

Provide row: exit
```

Chapter 4

Program structure

In this lab, you will add error handling and plotting functionality to an existing program. You will begin from the relatively short program `batch_means.py` (see the folder "Code for labs" on the course site), which could use some improvement. You will find test data in the files `sampleX.csv`, for $X \in \{1, 2, 3, 4\}$, in the same folder.

4.1 Learning goals

- You should be able to follow recommended practices for writing readable code.
- You should be able to rewrite existing code to follow recommended practices.
- You should be able to use exceptions for error handling.
- You should be able to use a library to plot data.

4.2 Submission

The lab should be submitted as usual via PeerGrade.

Submit your solution in a file with the final program. Include comments explaining the changes you have made for each task.

Obs: in this lab, you will use the library `matplotlib`, and you can also use the library `numpy` if desired.

4.3 The initial program

The goal of this program is to read in a data file with (invented) data from several batches of measurements, taken from different points on the plane, and for each batch calculate the average of the measurements taken inside the unit circle. A point (x, y) in the plane is inside the unit circle if $x^2 + y^2 \leq 1$.

Measurements taken outside the unit circle should be ignored. The data file has four columns separated with commas: it is a so-called `csv` file (where "csv" stands for *comma-separated values*). The first number records which batch a measurement belongs to, while the second and third record the x- and y-coordinates where the measurement was taken, and the fourth number is the measurement itself.

For example, a data file could contain the following lines:

```
1, 0.1, 0.2, 73
1, 0.11, 0.1, 101
2, 0.23, -0.01, 17
2, 0.9, 0.82, 23
```

Here we have measurements from two batches, 1 and 2, and so the program will calculate two averages. Note that the last measurement is outside the unit circle.

4.4 Tasks

4.4.1 A: Better structure (3 points)

Your task is to use what you've learned from the document "Basic principles of programming" on the course website and the compendium chapter "Rules of thumb for program structure and good code" to make this code more readable, easier to reuse in other projects, and easier to debug. You should be able to break up the one function that is given into several smaller, more reusable functions.

Test your code with the file `sample1.csv` as well as your own test files.

4.4.1.1 Requirements

- Your version of the program should not remove any functionality from the given program (don't worry about any bugs yet).
- Your functions should be documented with docstrings.
- Identifiers should be descriptive.

4.4.1.2 Tips

- It doesn't matter if your code is longer than the original file.
- Make changes step by step, verifying that the program still works after every change. It is harder to make a large change without affecting functionality.
- Write in a comment how you have improved the program.

4.4.2 B: Error handling (2 points)

If you test the program on the test files `sample2.csv`, `sample3.csv`, and `sample4.csv`, you will discover that there are some issues with `batch_means.py`. For this task, you will add error handling (to your improved program from Task A), using `try-except` so that crashes are avoided and the averages are calculated as well as possible despite issues in the input file. You should not correct the test files, but rather change the program so that it can deal with bad input data.

4.4.2.1 Requirements

- It should be possible to analyze all the given test files without issues.
- If it is impossible to open a filename that the user has entered, the program should point this out to them with a human-readable message rather than by raising an exception.
- If it is not possible to interpret a line of input data, the program should print a warning and ignore the line.
- Note in comments which errors you corrected and how you did it.

4.4.3 C: Sorted batch-data (1 point)

As `batch_means.py` is currently implemented, it will print the averages for each batch in an order determined by the order of lines in the input file. Change the program so that the printed output is sorted, i.e., so that the average for batch 1 is printed before batch 2 and so on. In other words, instead of an execution of the program looking like:

```
Which csv file should be analyzed? sample3.csv

Batch    Average
3        2.0
1        87.0
2        17.0
```

it should look like:

```
Which csv file should be analyzed? sample3.csv

Batch    Average
1        87.0
2        17.0
3        2.0
```

Tip: you probably want to use the function `sorted`. Read more in Python's documentation. Alternatively, you can use your own sorting algorithm from lab 3 (but remember to include it in your submission if you do).

4.4.4 D: Plotting the values (4 points)

Read up on the module `matplotlib`: https://matplotlib.org/stable/tutorials/introductory/pyplot.html.

You can import the library as follows:

```
import matplotlib.pyplot as plt
```

Obs: for this to work, you may first need to install `matplotlib`. You can do this in Anaconda or use the Python package installer `pip`. For installation instructions, see https://matplotlib.org/stable/users/installing.

Any function from `matplotlib.pyplot` can now be accessed by writing `plt.` before its name. For example, to use the `plot(...)` function, you would write `plt.plot(...)`.

Add now the following function:

```
def plot_data(data,f):
    # here be code
```

This function should plot the data stored in the argument `data` using `matplotlib`. You should not filter out points outside the unit circle, but you should plot the circle itself along with the points. You do not need to plot any averages. The plotted data should then be saved as a PDF in a file `f.pdf` (where `f` is the second parameter to `plot_data`).

The function `plot_data` should then be called in the main program after the averages have been printed, so that an execution of the program could look like so:

```
Which csv file should be analyzed? sample4.csv
```

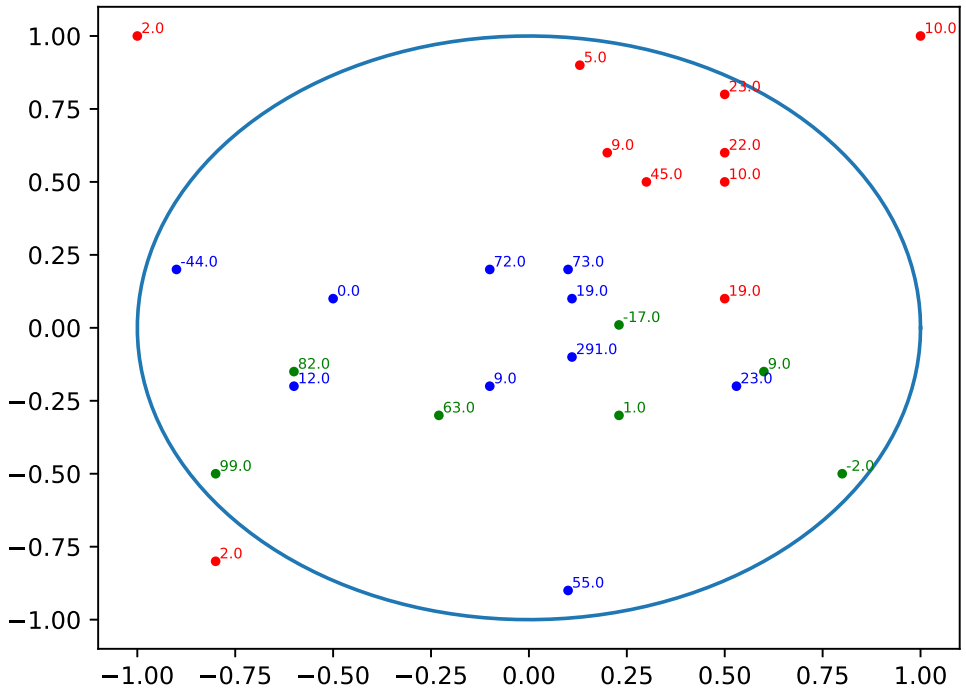
```
Warning: wrong input format for entry: 2  -0.93  -0.01  77

Warning: wrong input format for entry: 2  0.93  -0.01  53

Batch    Average
1        51.0
2        33.57142857142857
3        19.0

A plot of the data can be found in sample4.pdf
```

The file `sample4.pdf` would then contain the following plot:



Note that the unit circle looks more like an oval. This is due to the fact that the y-axis is smaller than the x-axis. It is possible to fix this, but you are not required to do so.

4.4.4.1 Requirements

- It should be possible to plot all the given test files without issues.
- All points, including those outside the unit circle, should be plotted. You do not need to plot any averages, only the measurements from the input file.

- All measurements in the same batch should be plotted with the same color, and each batch should have its own color. It is OK if you only support a finite number of differently-colored batches; the important thing is that it works for all the test files.
- You should draw the unit circle in the plot. There are many different ways to do so. The `numpy` library may be helpful.

4.4.4.2 Tips

- The type of the `data` parameter to your `plot_data` function will depend on how you solved the previous tasks. You should document its type in the function's docstring.
- For this task, you will find it useful to read documentation and look at examples from the internet.
- In this task, it is completely OK to use code you find online, but you should include a reference in a comment to the place you found the code.
- There are many different ways to draw a circle with `matplotlib`. Search the web for ideas! **Obs:** you are not required to use `numpy` to do this, but you can if you want to.

Chapter 5

Object-oriented programming

To introduce object-oriented programming, we will look at a computational problem from modern molecular biology. In DNA sequence analysis, we sometimes want to know how much overlap there is between two sequences (simplified: how much DNA they have in common). You will write simple object-oriented code to support such an analysis.

5.1 Learning goals

- You should be able to write your own class, and methods belonging to that class, in order to solve programming problems.
- You should be able to use functions as arguments to functions.
- You should be able to use exceptions to handle errors.

5.2 Submitting your code

A code file with your solutions is to be uploaded to PeerGrade. Submit *one* Python file that solves the tasks.

5.3 Given code

There is a code stub `dna.py` in the lab code directory for the course. This file contains a starting point for your code (quite minimal) plus some test functions. The test functions have two purposes. (1) They make your testing easier and faster. It is often convenient to have some basic testing available. Later, we will look at "real" systematic testing. (2) The test is a codified specification of how your functions should work. You are done when the call `test_all()` prints `Yay, all good`.

5.3.1 Tasks

5.3.2 Define a class for DNA sequences (2 points)

Define the class DnaSeq to hold two string attributes: an accession (which is supposed to be a unique identifier of a DNA sequence) and a DNA sequence. Name the attributes `accession` and `seq`. It should be possible to define a DnaSeq object like this:

```
my_seq = DnaSeq('abc123', 'ACGTACGT')
```

Requirements

- Your class should have a constructor (`__init__`, as given in the stub) and two methods: `__len__` and `__str__`. Methods with these names have special meaning in Python: if a class has these two methods defined, then one can apply the functions `len` and `str` to them.
- In this exercise, `__len__` should return the length of the DNA sequence (so ignore the accession), and `__str__` should return a "label" for convenient printing looking like

  ```
  <DnaSeq accession='abc123'>
  ```

  when applied to the example above. That is, you should use a template like `<DnaSeq accession='XXX'>` and insert the accession of the object instead of `XXX`.
- Instantiation with empty accession or sequence strings should raise a ValueError exception. That is, a call like `DnaSeq('', None)` should raise the exception.
- The methods should not print anything, just return values.
- When you are done, the call `test_class_DnaSeq()` should print `DnaSeq passed`, and no errors should occur.

5.3.3 Reading DNA sequences from file (3 points)

Implement the function `read_dna()`, which given a filename as parameter returns a list of `DnaSeq` objects representing the sequences in the file. The file format is as follows:

- A file is a sequence of records.
- Each record consists of an *accession line* and a *sequence line*.
- The accession is a unique identifier, represented by a string, and the accession is preceeded by a greater-than sign (>) which is not part of the accession.
- The sequence line is a string of A, C, G, and T.
- There may be blank lines in between the records. They should be ignored. Note that blank lines do not have length 0, because the newline character is part of the input.

Here is a simple example file:

```
>s123
ACGGACGT
>abc
GATTACA

>X20
AAAAAAGAATTACCCACACACAC
```

The data folder on the course web site contains files (suffix ".fa", these are regular textfiles, have a look!) which you can use to test your code on. The files `ex1.fa`, `ex2.fa`, and `sars_cov_2.fa` are used by the tests, so you should download them.

Requirements

- The call `test_reading()` should print `read_dna passed`, and no errors should occur.
- Newline characters need to be removed from accessions and sequences.
- The function should have a docstring.

5.3.4 A function to check overlaps (2 points)

We want to be able to detect *overlaps* between sequence using a function called `check_exact_overlap`. We say that sequence a overlaps with sequence b if there is a suffix in a of some length L that is identical to a prefix of length L in b. Note that this definition is not commutative: a can overlap with b, without b overlapping with a.

Example 1: The DNA sequence AAACCC has an overlap of length 3 with CCCCGATT, because the last three characters of AAACCC are identical with the first three characters of CCCCGATT. These sequences have no overlap of lengths 4 or longer.

Example 2: AAACCC has no overlap with GTACCC or GGGCCCTTT, because there is no suffix of AAACCC that is identical to a prefix of GTACCC or GGGCCCTTT.

Example 3: AAACCC overlaps with ACCC. In fact, it overlaps completely, with length 4.

Implement `check_exact_overlap` to take three parameters: two DnaSeq objects (corresponding to a and b above) and a minimum length. Overlaps shorter than the minimum length are to be ignored.

Requirements:

- The minimum overlap length parameter should have a default value of 10.
- The length of the longest detectable overlap larger than the minimum length should be returned.
- Return 0 if no overlap is detected.
- The function should have a docstring.

5.3.5 A higher-order function for checking all overlaps (3 points)

Implement the function `overlaps` which takes two parameters:

- A list of DnaSeq objects.
- An overlap detection function.

An example of an overlap detection function is `check_exact_overlap`, but any replacement would do. In the testing code, alternative overlap functions are constructed from `check_exact_overlap` using other minimum overlap lengths. If `lst` is a list of DnaSeq objects, then `overlaps(lst, check_exact_overlap)` is a valid call to the function.

All detectable overlaps among pairs of sequences in the input list are to be returned.

Requirements:

- Return a dictionary of dictionaries containing the lengths of overlaps between sequences. If `d` is the result of a call to `overlaps`, and the sequences with accessions `s1` and `s2` overlaps with length 10, then `d['s1']['s2'] == 10` should be true. If those two sequences were the only input, the nested dictionary could be written as `{'s1': {'s2': 10}}`.
- There should be no empty entries in the returned dictionary. If `s1` does not overlap with any of the input sequences, then there should not be an entry for `s1` in the returned dictionary.
- Nothing should be printed, only returned, by the function.
- The function should have a docstring.
- The call `test_overlap()` should print `overlap code passed` and no errors should occur.

5.4 Notes for the interested

- The format we have worked with in this lab is a simplified version of the Fasta format, hence the file suffix ".fa".
- We look at DNA data as if we know the direction. That is in practice often not the case. One has to consider both forward and backward direction of each sequence, and with respect to so-called reverse complement of DNA.
- The type of overlaps we are looking at is the core of *genome assembly*, but one cannot work with perfect overlaps (real sequences have errors) as we do, and one has to use faster algorithms than we are considering since the typical data size is so much larger.