IMoL Python course notes

Ditlev E. Brodersen & Mads-Peter V. Christiansen

Contents

1 Introduktion til Livets Molekyler 5

2 TØ 1. Getting started 7
2.1 Programming languages 7
2.2 Python as a calculator 7
2.3 Calculations & Parentheses 8

3 TØ 3. Variables & Functions 9
3.1 Variables 9
3.2 Functions 10

4 TØ 3. Extra 13
4.1 Functions 13

5 TØ 4. Laboratory Exercise 1 15
5.1 Pipette measurements 15
5.2 Adsorption measurements 16

6 TØ 4. Extra 17
6.1 Mean and standard deviation 17

7 TØ 5. Data types 19
7.1 The primary data types. 19
7.2 Data collections: Lists 20
7.3 Data collections: Revisiting strings 22

8 TØ 5. Extra 25
8.1 Data collections: Arrays 25
8.2 Data collections: DataFrames 25

9 TØ 7. Working with data 29
9.1 Plotting 29

10 TØ 7. Extra 31
10.1 Data manipulations 31

11 TØ 9. Laboratory Exercise 2 33
11.1 Enzyme Kinetics 33

1. Introduktion til Livets Molekyler

On this site you will find the exercises related to Python for the *Introduktion til Livets Molekyler* course. Use the sidebar or the table below to navigate the site.

For each page you can download it as either PDF or docx from the menu on the rightside - alternatively you can click here to download a combined version as pdf.

2. TØ 1. Getting started

Python is a programming language that is widely used in the scientific community (and for many other things). You will be using Python in various contexts during your studies - ranging from data analysis through plotting to bioinformatics.

At first Python may seem strange and difficult - it is a different way to think compared to what you might be used to. However, with practice and patience you will become familiar with it and be able to use it to learn and do science in ways that would not be possible without it.

2.1 Programming languages

You have almost certainly used something that works like a programming language before — you may simply not have thought about it that way. For example, a calculator uses a programming language of sorts - it has a set of inputs (numbers) and a set of operations (add, substract, multiply, divide, etc) that can be composed together to perform a calculation. Python can do the same, and much more, for example

```
1 + 2 + 3
```

> ! Important
>
> This is an interactive code-block that runs Python in the browser. You can click the **Run Code** button to perform the calculation.

2.1.1 Exercise: Change the calculation.

You can edit the code in the cell above just click on the the line with code and you can start typing. Try for example calculating $1 + 2 + 3 + 4$ rather than the original calculation.

2.1.2 Exercise: Other programming languages

If we define a programming language as

> A programming language is a way of writing precise instructions that tell a computer what operations to perform and in what order.

Think of other systems you have used that might qualify as a programming language. They should allow you to combine operations or instructions to produce a result.

2.2 Python as a calculator

All operations you're used to for a regular handheld calculator exist in Python

```
1 + 1       # Addition with +
3 - 1       # Subtraction with -
4 * 4       # Multiplication with *
20 / 5      # Division with /
```

2.2.1 Exercise: Calculate, calculate, calculate

Use the interactive cell below to perform the following calculations

1. $21 + 21$

2. $53 - 11$
3. 6×7
4. $\frac{546}{13}$

______

2.3 Calculations & Parentheses

In later courses you will need to work with equations such as the one shown below

$$\theta = \frac{K_D + [P_{tot}] + [L_{tot}]}{2[P_{tot}]} - \sqrt{\left(\frac{K_D + [P_{tot}] + [L_{tot}]}{2[P_{tot}]}\right)^2 - \frac{[L_{tot}]}{[P_{tot}]}}, \tag{2.1}$$

which can be used to describe titration data that reports protein saturation as a function of total ligand concentration.

Calculating this function requires that we are careful with parentheses! Which we have all learned, but since it is very important it is worth refreshing.

2.3.1 Exercise: Mind the (...)

For the following pairs of expressions, calculate both (by hand or mentally, according to preference.)

$$\begin{aligned} 2 + 3 \times 4 \quad &\text{vs.} \quad (2+3) \times 4 \\ 10 - 2^2 \quad &\text{vs.} \quad (10-2)^2 \\ 100/10 \times 2 \quad &\text{vs.} \quad 100/(10 \times 2) \\ \frac{12+8}{4} \quad &\text{vs.} \quad 12 + 8/4 \end{aligned} \tag{2.2}$$

2.3.2 Exercise: Beware of the (...)

For each of the interactive cells below, consider what the result will be **before** you run the cell.

```
a = 1 + 2 * 3 ** 2
print('a: ', a)
```

```
b = (1 + 2) * 3 ** 2
print('b: ', b)
```

```
c = 1 + 2 * (3 ** 2)
print('c: ', c)
```

3. TØ 3. Variables & Functions

In this session we will explore variables and functions - both of which are important elements of Python (and many other programming languages)

3.1 Variables

To really unlock the utility of Python variables are required, among other things they allow us to express calculations in a simpler way. A variable is defined by using `=`, for example, we can define

```
a = 21 + 21
```

This means that the result of the calculation is stored in the variable `a` and can be used in subsequent calculations.

3.1.1 Exercise: Calculations with variables

Repeat the calculations

1. $21 + 21$
2. $53 - 11$
3. 6×7
4. $\frac{546}{13}$

But now assign each of them to a variable `a`, `b` `c` etc and then calculate

$$(21 + 21) + (53 - 11) + (6 \times 7) + \frac{546}{13} \quad (3.1)$$

using these variables. In the cell below replace ______ with your code

```
a = ______
b = ______
c = ______
d = ______
f = ______ # Do not use any numbers on this line.

print(a, b, c, d, f)
```

i Note

The cell above used a new thing `print`. This is a *function*, functions are like recipe - they take a number of inputs (ingredients) and perform some predefined set of operations on those and return an output.

The `print`-function prints what it is given to the screen - in this case the five variables. You will see and use the `print`-function numerous times.

3.1.2 Exercise: A biological calculation

The number of cells in different parts of the body is given below

Type	Number of cells
Total cells	$37 \cdot 10^{12}$
Brain cells	$86 \cdot 10^{9}$

Type	Number of cells
Liver cells	$240 \cdot 10^9$
Skin cells	$16 \cdot 10^{11}$

For each cell type, calculate the percentage each category represents of the total number of cells in the body, i.e., for each category calculate

$$P = \frac{\text{Number}}{\text{Total}} \times 100. \tag{3.2}$$

```
# Make variables for the different number of cells
total_cells = ______
brain_cells = ______
liver_cells = ______
skin_cells = ______

# Calculate the percentages - use the variables!
brain_pct = ______
liver_pct = ______
skin_pct = ______

# Print the results
print("Brain cell percentage: ", brain_pct)
print("Liver cell percentage: ", liver_pct)
print("Skin cell percentage: ", skin_pct)
```

Tip

The numbers of cells are rather large, so typing the correct amount of zeros is a little painful. The code below shows four different ways of writing the same number in Python code

```
13000000000000     # Basic
13_000_000_000_000 # Underscores to visually distinguish
13 * 10**(12)      # Using exponentation
13E12              # Scientific notation
```

3.2 Functions

When doing repeated calculations it is generally good practice to define a function. Consider for example that you need to calculate the average of several sets of numbers, without using functions we could write

```
A = 1
B = 2
C = 3
D = 4

average_AB = (A + B) / 2
average_BC = (B + C) / 2
average_CD = (C + D) / 2
```

This is perfectly valid, but it is more prone to error - as we have to write the expression multiple times - and it is harder for someone else to decipher - as they have to check they understand what each line of code does. If we instead used a function

```
A = 1
B = 2
C = 3
```

```
D = 4

def average(x, y):
  return (x + y) / 2

average_AB = average(A, B)
average_BC = average(B, C)
average_CD = average(D, C)
```

Now all the logic of calculating the average is contained in the `average` function so we only have one place to check for errors.

i A note on functions

Functions use specific syntax, meaning they have to be written following a set of rules

```
def percentage(number, total):
  intermediate_calculation = number / total
  final_result = intermediate_calculation * 100
  return final_result
```

Line 1
The function is defined with `def` and its name (here `percentage`) and its inputs `(number, total)`.

Line 2
The body of the function is indented by 4 spaces (or 1 tab). The body can consist of as many lines as needed.

Line 4
`return` exits the function and returns the variable that follows - here `final_result`. If a function doesn't have a `return` or just has `return` without any variables it implicitly returns `None`.

Using a function requires "calling" it, this means giving it the inputs it expects - like below

```
result = percentage(1, 2)
```

This code will calculate the function with `number=1` and `total=2` and assign the returned value to `result`. Outside the function the variables `intermediate_calculation` and `final_result` are not available (we would say they are out of scope). There's also nothing special about the names, they are just variables like anywhere else, they could just aswell have been called `nemo_the_fish` or `min_endelige_beregning_af_gennemsnittet_foretaget_mandag_den_13_april` but its a good idea to give variables descriptive but short names.

Only what was returned by the function is available outside the function and if it needs to used later on in the code it needs to be saved to its own variable, so we can think of the code above like

1. `percentage(1, 2)` calculates and returns `final_result`.
2. Whatever is returned is assigned to the variable `result`.

3.2.1 Exercise: A biological calculation with a function.

Repeat the calculation of cell percentages but now using the `percentage` function defined below

```
# Define a function to calculate the percentage
def percentage(number, total):
  # This function is equivalent to the one shown above, just a little more concise.
  return number / total * 100

# Make variables for the different number of cells
total_cells = 37_000_000_000_000
brain_cells = 86_000_000_000
liver_cells = 240_000_000_000
skin_cells = 1_600_000_000_000

# Calculate the percentages - use the variables!
brain_pct = ______
```

```
liver_pct = ______
skin_pct = ______

# Print the results
print("Brain cell percentage: ", brain_pct)
print("Liver cell percentage: ", liver_pct)
print("Skin cell percentage: ", skin_pct)
```

3.2.2 Exercise: Fix the (...)

We want to calculate

$$\frac{a+b}{c+d} \tag{3.3}$$

Someone has implemented the function below to do so, but it is incorrect - your task is to fix it.

```
def fraction_func(a, b, c, d):
    return a + b / c + d

result = fraction_func(152, 16, 2, 2)
print(result)
```

3.2.3 Implement the (...)

The first term of the quadratic binding equation[1] is

$$\frac{K_D + [P_{tot}] + [L_{tot}]}{2[P_{tot}]} = \frac{a+b+c}{2b} \tag{3.4}$$

Complete the function below so it calculates this expression

```
def first_term(a, b, c):
    return ______

result = first_term(100, 2, 66)
print(result)
```

[1]In many programming languages, counting starts at 0 instead of 1. This comes from how computers store lists in memory. You can think of a list as starting at some location in memory, and each new element being a fixed step away.
The index then tells the computer how many steps to move from the start:

- index `0` ⟶ move 0 steps ⟶ first element
- index `1` ⟶ move 1 step ⟶ second element
- index `2` ⟶ move 2 steps ⟶ third element

This makes it simple and efficient for the computer to find elements. Most modern programming languages have kept this convention.

4. TØ 3. Extra

This note contains additional exercises regarding functions and variables.

4.1 Functions

Functions in Python are like recipes in a cookbook. Writing them down does not “run” them, just like how writing down a recipe for a delicious brownie does not cause a delicous brownie to appear.

To “run” a function we need to *call* it, for example

```
def my_func(x):
    result = 2 * x
    return result

output = my_func(2)
```

Line 1
  Function definition, does not run the function.

Line 5
  Function call, runs the functions for the input `x=2`.

Additionally, if we want the function to influence our program we need to assign its output to a variable[2]. Above we assigned the output of the function to the variable `output`, so we now have access to that variable

```
print("The output was: ", output)
```

```
The output was:  4
```

Then there is the concept of scope, the function we defined above defines a variable `result`, lets see what happens if we try to access it

```
print(result)
```

```
NameError: name 'result' is not defined
□[31m---------------------------------------------------------------------------□
[39m
□[31mNameError□[39m                                 Traceback (most recent call last)
□[36mCell□[39m□[36m □[39m□[32mIn[3]□[39m□[32m, line 1□[39m
□[32m----> □[39m□[32m1□[39m print(result)

□[31mNameError□[39m: name 'result' is not defined
```

Python throws an error, telling us that `result` is not defined. This is because the `result`-variable is only available in the *scope* of the function. That means `result` only exists inside the function.

[2]In many programming languages, counting starts at 0 instead of 1. This comes from how computers store lists in memory. You can think of a list as starting at some location in memory, and each new element being a fixed step away.
The index then tells the computer how many steps to move from the start:

- index `0` ⟶ move 0 steps ⟶ first element
- index `1` ⟶ move 1 step ⟶ second element
- index `2` ⟶ move 2 steps ⟶ third element

This makes it simple and efficient for the computer to find elements. Most modern programming languages have kept this convention.

4.1.1 Exercise: Functions, calling, assignment and scopes

```
# A function that calculates something but doesn't return it
def function_without_return(A, B):
  result = A * B
  print("Inside function_without_return: ", result)

# A function that calculates something and returns it.
def function_with_return(A, B):
  result = A * B
  print("Inside function_with_return: ", result)
  return result
```

The code block above defines two functions.

Below there are three code blocks that use these functions and an accompanying question. Use the interactive codeblock below to answer these questions.

You are not expected to give technically precise answers, just describe it in your own words. For each question you can play around with the interactive block or the definition of the functions above to help you answer.

4.1.1.1 Without assignment

What happens when the functions are called but the output not assigned?

```
# Call the functions but don't assign to a variable
print('Calling functions without assigning to a variable')
function_without_return(6, 7)
function_with_return(6, 7)

print('End of cell')
```

4.1.1.2 With assignment

What happens when the functions are called and the output is assigned?

```
print('Calling functions and assigning')
without_return = function_without_return(6, 7)
with_return = function_with_return(6, 7)

print('without_return: ', without_return)
print('with_return: ', with_return)
print('End of cell')
```

4.1.1.3 Without calling

What happens when you assign the function itself to a variable without calling it?

```
print('Assigning to a variable but not actually calling')
without_return_no_call = function_without_return
with_return_no_call = function_with_return

print('without_return_no_call: ', without_return_no_call)
print('with_return_no_call: ', with_return_no_call)
print('End of cell')
```

5. TØ 4. Laboratory Exercise 1

In this session we will use Python to analysis data gathered in the laboratory. First we will start by using Python to calculate statistics of pipette measurements and afterwards we will use Python calculate and plot the standard curve for absorption measurements.

5.1 Pipette measurements

In the laboratory, you carried out a pipetting exercise in which you had to measure the same volume three times in order to assess your precision (Exercise 1, protocol step 12).

In this session we will stary by using Python to calculate the mean and standard deviation of these results.

5.1.1 Exercise: A function for the mean

The mean of three numbers x_1, x_2, x_3 can be written as

$$m(x_1, x_2, x_3) = \frac{x_1 + x_2 + x_3}{3} \tag{5.1}$$

In the cell below finish implementing the function `mean_func` for calculating the mean

```
def mean_func(x1, x2, x3):
  result = ______
  return result

# Its always a good idea to experiment
# Consider what the result of the below should be
test = mean_func(1, 2, 3)
print(test)
```

5.1.2 Exercise: Calculating the mean and standard deviation

In the cell below the two functions `mean_func` and `std_func` are available.

Start by setting the variables `x1`, `x2`, `x3` equal to your measurements, then use the two functions to calculate the mean and the standard deviation.

```
# Define your measurements
x1 = ______
x2 = ______
x3 = ______

# Calculate the mean and standard deviation
mean_value = mean_func(_____, _____, _____)
std_value = ______ # Use std_func
```

If you want to try writing a more general function for calculating the mean see the extra exercises.

5.1.3 Exercise: Analyzing the standard deviation

What does the standard deviation tell you about your precision?

If the standard deviation is high, what could this be due to in your pipetting?

5.2 Adsorption measurements

In the laboratory you made adsorption measurements, now you will use Python to make a standard curve from these measurements. Start by uploading your measurements

```
//| echo: false
viewof xlsx_file = Inputs.file({
  label: "Upload Excel file",
  accept: ".xlsx",
  required: false
})

xlsx_name = xlsx_file ? xlsx_file.name : null
xlsx_bytes = xlsx_file
  ? Array.from(new Uint8Array(await xlsx_file.arrayBuffer()))
  : null
```

This loads the data as a `pd.DataFrame` which will be touched on later and used extensively in other courses.

The next cell calculates the mean and the standard deviation across the replicates

```
mean = df[["m1", "m2", "m3"]].mean(axis=1)
std = df[["m1", "m2", "m3"]].mean(axis=1)

pd.DataFrame({"conc": df["conc"], "mean": mean, "std": std})
```

And now we can plot it the standard curve

```
# Create the plotting window
fig, ax = plt.subplots()

# Plot mean with std errorbars.
ax.errorbar(df['conc'], mean, yerr=std,
  label="Mean ± SD", fmt="o-", capsize=5)

# Plot customization
ax.set_xlabel("Concentration (µM)")
ax.set_ylabel("Absorption (A400)")
ax.set_title("Average adsorption vs concentration")
ax.legend()

# Display the plot
plt.show()
```

5.2.1 Exercise: Interpretation

How can you tell whether you have made your dilutions and carried out your absorbance measurements with high accuracy? What can a standard curve be used for?

6. TØ 4. Extra

This note contains additional exercises related to laboratory exercise 1.

6.1 Mean and standard deviation

For the pipette exercise we implemented a function to calcualte the mean of three values.

The exercise belows explores how to create a more general function that calculates the mean of any number of values.

6.1.1 Exercise: General mean function

A more general equation for the mean is

$$m(X) = \frac{1}{\mid X \mid} \sum_{x \in X} x \tag{6.1}$$

Where X is a set of numbers, $\mid X \mid$ is the size of the set i.e. the amount of numbers. The function we implemented before would only take the average of three numbers, while that is useful it would be better to have a more general function.

To build this function we will need a few changes

1. We need the input to the function to be an arbitrary number of numbers.
2. We need to sum over all the input numbers.
3. We need to divide by the number of inputs numbers.

Luckily, all of this is very achievable in Python.

1. We will use a list of numbers `[1, 2, 3, 4, ...]` as input.
2. We will use the `sum`-function to calculate the sum
3. We will use the `len`-function to find the length of the input list.

Finish the code in the cell below

```
def mean_func(numbers):
  sum_total = ______ # Use `sum`
  amount = ______ # Use `len`
  mean = ______ # Divide sum_total by amount
  return mean

test_input = [1, 2, 3, 4, 5]
print(mean_func(test_input))
```

7. TØ 5. Data types

One of the most useful aspects of Python is the ability to work with complex datasets. However, to do so it is useful to understand a little bit about **data types**.

In the end a computer stores everything as a binary sequence e.g. `0101001` but Python (and many other programming langauges) add abstractions that makes it easier to reason about our programs. We will start by exploring some primary data types and then look at a simple collection type, the list.

In the extra material we explore other types of collections, namely arrays and dataframes.

7.1 The primary data types.

We will consider three primary data types

- **Integers** (*whole numbers*): Used to represent whole numbers, for example the number of bases or codons.
- **Floats** (*decimal numbers*): Used to represent decimal numbers, for example a concentration in $\frac{\text{ng}}{\mu\text{L}}$.
- **Strings** (*text*): Used to represent text, for example mRNA-sequences or names.

The code block below shows a variable of each of these types

```
an_integer = 1
a_float = 3.1415
a_string = "Eukaryote"

print('Integer: ', an_integer, type(an_integer))
print('Float: ', a_float, type(a_float))
print('a_string: ', a_string, type(a_string))
```

> Tip
>
> Here the function `type` was used to obtain the type that Python uses for each of our variables. You don't need to understand the details of this - just notice that these three pieces of data were automatically understood to be one of the above types.

7.1.1 Exercise: mRNA and datatypes.

Consider working with an mRNA molecule with sequence `GCAUGCCGUUUGGAUGACCG`.

7.1.1.1 Sequence data type

What Python data type would it be natural to use for an mRNA sequence?

7.1.1.2 Sequence length

What Python data type would it be natural to use for the length of an mRNA sequence?

7.1.1.3 In Python

Using the cell below, how many nucleotides are in the sequence?

```
mrna = "GCAUGCCGUUUGGAUGACCG"
sequence_length = len(mrna)

print("The sequence has length:", sequence_length)
```

> Tip
>
> The `len` function is incredibly useful - here we used it to return the number of characters in a string.

7.1.1.4 mRNA codons

Use the cell the below to calculate the number of codons in the mRNA

```
mrna = "GCAUGCCGUUUGGAUGACCG"
sequence_length = len(mrna)
number_of_codons = ______

print("Number of codons:", number_of_codons)
```

> Division and floats
>
> You may have notice that the number printed above is a float not an integer.
> This is because Python converts division of integers into floats, for example
>
> ```
> numerator = 10
> denominator = 5
> print(numerator / denominator)
> ```
>
> ```
> 2.0
> ```
>
> If we know that we want an integer we can either convert back to int or use a different division operation
>
> ```
> numerator = 10
> denominator = 5
> print(numerator // denominator)
> ```
>
> ```
> 2
> ```

7.2 Data collections: Lists

The three primary types we saw above can describe a lot of things, however an actual dataset is typically a collection of several measurements/labels/etc. Imagine for example an experiment where the absorption coefficent is measured as a function of wavelength, we could represent that as

```
wavelength_1 = 400
absorption_1 = 2.451
wavelength_2 = 401
absorption_2 = 2.532
...
```

However, its pretty clear that this becomes really annoying to deal with very quickly - so we need something smarter. The simplest collection is a list

```
wavelengths = [400, 401, 402, 403, 404, 405]
absorptions = [2.451, 2.532, 2.567, 2.621, 2.584]
```

This is not too bad, it keeps the wavelengths grouped together and similarly for the absorptions.

i Using lists for plotting

Lists are also the type of input that is usually used for plotting, consider for example the cell below that makes a very basic plot.

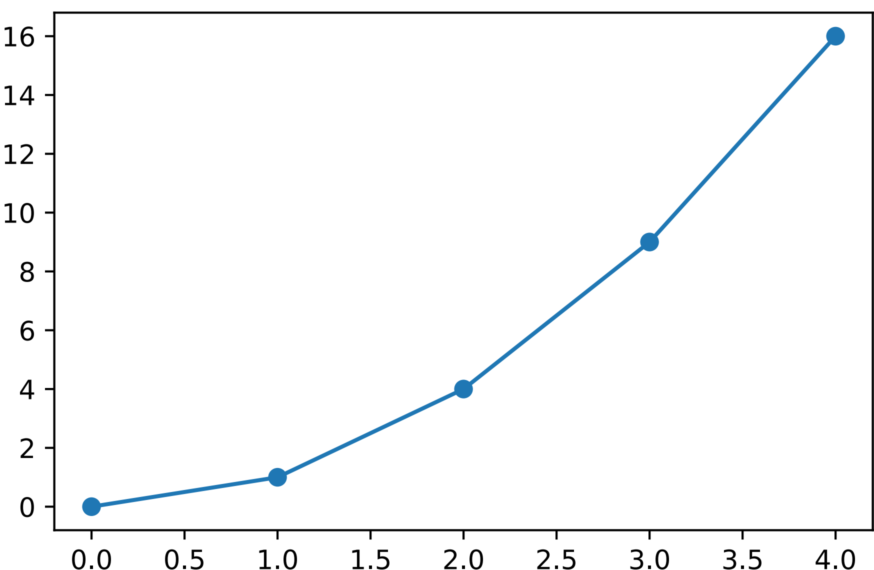
```
import matplotlib.pyplot as plt

x = [0, 1, 2, 3, 4]
y = [0, 1, 4, 9, 16]

fig, ax = plt.subplots()

ax.plot(x, y, '-o')
```

Line 3
Make a list of x-coordinates.

Line 4
Make a list of y-coordinates.

Line 6
Make the figure (fig) and axis (ax) for plotting

Line 8
Plot the data in the axis.



This code produces the shown figure. You will learn much more about plotting through out your studies as it is an essential part of working with scientific data.

For plotting we use every entry in the lists, but there are many situations in which we want to select a specific element of a dataset - this is achieved through *indexing*.

For example with the `wavelengths`-list we made before, we can index elements like so

```
element_1 = wavelengths[0]
print("The first element is: ", element_1)
```

```
The first element is:  400
```

The indexing happened with the code `wavelengths[0]` where `[0]` specifies we want the first element. This may seem a little surprising, why do we use `[0]`? There is a simple reason for this, explained briefly in the footnote[3].

[3] In many programming languages, counting starts at 0 instead of 1. This comes from how computers store lists in memory. You can think of a list as starting at some location in memory, and each new element being a fixed step away.
The index then tells the computer how many steps to move from the start:

7.2.1 Exercise: Indexing a list of molecules

In an experiemnt, several molecules were identified in a sample, use the cell below to

1. Print the first molecule
2. Print the third molecule
3. Print the last molecule

```
molecules = ["glucose", "ATP", "cholesterol", "lactate", "glycine"]

first_index = ______
print("The first molecule is: ", molecules[first_index])

third_index = ______
print("The third molecule is: ", molecules[third_index])

last_index = ______
print("The last molecule is: ", molecules[last_index])
```

7.2.2 Exercise: Building a sentence

The following list contains parts of a sentence in the wrong order.
Fill in the indices so that the printed sentence is correct.

```
sentence_pieces = [
    "powerhouse",
    "are the",
    "mitochondria",
    "of the cell"]

first_index = ______
second_index = ______
third_index = ______
fourth_index = ______


print(
    sentence_pieces[first_index],
    sentence_pieces[second_index],
    sentence_pieces[third_index],
    sentence_pieces[fourth_index]
)
```

7.3 Data collections: Revisiting strings

Earlier we introduced strings as a basic datatype, but in some ways they can actually also be considered a collection - and therefore strings can also be indexed.
For example, we can print the fourth element of an mRNA sequence like shown below

```
mrna = "GCAUGCCGUUUGGAUGACCG"
print(mrna[3])
```

```
U
```

This also lets us use another form of indexing a *slice*, for example

- index `0` ⟶ move 0 steps ⟶ first element
- index `1` ⟶ move 1 step ⟶ second element
- index `2` ⟶ move 2 steps ⟶ third element

This makes it simple and efficient for the computer to find elements. Most modern programming languages have kept this convention.

```
mrna = "GCAUGCCGUUUGGAUGACCG"
codon = mrna[0:3]
print(codon)
```

```
GCA
```

Here `[0:3]` means: start at index 0 and stop *before* index 3.

7.3.1 Exercise: Find the third codon.

For the `mrna` sequence use a slice to print the third codon

```
mrna = "GCAUGCCGUUUGGAUGACCG"
codon = mrna[______]
print(codon)
```

8. TØ 5. Extra

This note describes arrays and DataFrames - both of which are collection types that offer additional features compared to lists that are very useful for scientific data.

8.1 Data collections: Arrays

Another very common collection is an *array*. Arrays are typically used for numerical data as they make computations simple as they enable *element-wise calculations*. This means that we can perform the same calculation for every element in an array.

Consider for example converting distances in kilometers to distances in meters.

```
import numpy as np

distance_in_km = np.array([1.43, 75.12, 9.042, 1.337])
distance_in_m = distance_in_km * 1000
print(distance_in_m)
```

```
[ 1430. 75120.  9042.  1337.]
```

This works with all common arithmetic operations

- Addition: `my_array + 10`
- Subtraction: `my_array - 10`
- Multiplication: `my_array * 10`
- Division: `my_array / 10`

8.1.1 Exercise: Concentration conversion

In an experiment you have measured the concentration a solution in units of μM (micromolar) but you need to work with them in millimolar. Use the cell below to convert the concentrations.

```

import numpy as np

conc_uM = np.array([137.03, 205.1, 56.4, 732.2, 1563.3])

conc_mM = ______

print("Concentrations in millimolar: ", conc_mM)
```

8.2 Data collections: DataFrames

Lists lets us avoid having a huge amount of variables, arrays extend that to also let us make calculations across the entire dataset - however we are still left with only an implicit connection between different arrays.

We can go one step further and work with `DataFrames` which are kind of an extension of an excel-document in Python.

```
import pandas as pd

df = pd.DataFrame({'wavelengths': [400, 401], 'adsorptions': [2.451, 2.532]})
print(df)
```

```
   wavelengths  adsorptions
0          400        2.451
1          401        2.532
```

Now the full dataset is contained in one variable (here called `df`) - this makes working with a dataset much easier. A `DataFrame` can also be indexed, for example we extract the `wavelengths` like so

```
df['wavelengths']
```

```
0    400
1    401
Name: wavelengths, dtype: int64
```

So with a `DataFrame` indexing uses the name of the row or column rather than the numeric index we've seen for lists and arrays.

8.2.1 Exercise: Calculating cell sizes

The code below creates a dataset using a `DataFrame` with cell types and the upper and lower limits of their radii.

```
import pandas as pd

cell_data = pd.DataFrame({
    'cell_type': ["Mycoplasma", "Typical bacterium", "Yeast cell", "Red blood cell",
   "Lymphocyte", "Typical animal cell", "Typical plant cell", "Human egg cell (oocyte)",
    "Neuron (cell body)"],
'radius_lower_um': [0.2, 1, 5, 7, 8, 10, 20, 120, 10],
'radius_upper_um': [0.3, 2, 7, 8, 10, 20, 100, 140, 100]}).set_index('cell_type')

print(cell_data)
```

We would like to calculate the volume of these cells, assuming they are spherical

$$V(r) = \frac{4}{3}\pi r^3 \tag{8.1}$$

8.2.1.1 Calculate the volume of a myocoplasma

We will start by making the calculation as if Python was almost just a normal calculator.

Put in the correct lower limit of the radius and calculate the volume of the cell in the interactive cell below.

Here you are not expected to make use the `cell_data` variable.

```
pi = 3.1415
mycoplasma_radius = ______ # Write the correct radius
mycoplasma_volume = ______ # Calculate the volume

print("The volume is", mycoplasma_volume)
```

8.2.1.2 Calculate the volume of mycoplasma - but smarter.

It's kind of silly to create the `cell_data`-table (or rather `DataFrame`) and then not use it. So now we want to extract the lower limit of the radius of the mycoplasma cell from the `DataFrame` rather than writing it manually.

To do so we need to use *indexing* to extract the entry we are interested in.

```
pi = 3.1415
mycoplasma_radius = cell_data['radius_lower_um'][______]
mycoplasma_volume = ______
```

```
print("The volume is", mycoplasma_volume)
```

8.2.1.3 Calculate the volume of all the cells

Each column in `cell_data` is actually an array, so we can easily calculate volumes of every cell at once.

```
pi = 3.1415

# This extracts the array of lower limit cell sizes.
cell_volumes = ______ * cell_data['radius_lower_um']**3

print(cell_volumes)
```

9. TØ 7. Working with data

In this note we will look at how to plot data from a dataset. We will be using the dataset presented in this paper on "the absorption characteristics of extracted phytoplankton pigments".

We use a `DataFrame`, a collection type that is similar to a table/excel-sheet, to store the data. If you are interested in learning more about these you can look at the extra material for TØ5. This weeks extra material dives into how to do calculations with a `DataFrame`.

The table below shows an excerpt of the dataset. The first column `wavelength_nm` is the wavelength in nanometers while the other columns are adsorptions measurements for different substances.

```
import pandas as pd

pigments = pd.read_csv('../../datasets/curated_abs_dataset.csv')
display(pigments)
```

	wavelength_nm	chlorophyll-a	chlorophyll-b	fucoxanthin	prasinoxanthin
0	800.2	0.0	0.000035	0.000629	0.000128
1	799.3	0.0	0.000000	0.000588	0.000166
2	798.5	0.0	0.000000	0.000599	0.000165
3	797.6	0.0	0.000000	0.000548	0.000216
4	796.7	0.0	0.000086	0.000491	0.000086
...	...	...	...	...	...
558	324.0	NaN	NaN	0.003124	0.002171
559	323.1	NaN	NaN	0.003056	0.002062
560	322.3	NaN	NaN	0.003098	0.002207
561	321.4	NaN	NaN	0.003092	0.002202
562	320.6	NaN	NaN	0.003042	0.002189

9.1 Plotting

One of the best ways to explore a dataset is to plot it, below one column of the dataset is plotted

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

# Plotting
ax.plot(pigments['wavelength_nm'],
      pigments['chlorophyll-a'],
      label='Chlorophyll-a')

# These are settings to make the plot look nice
ax.set_xlabel('Wavelength [nm]')
ax.set_ylabel('Adsorption')

ax.legend()
```

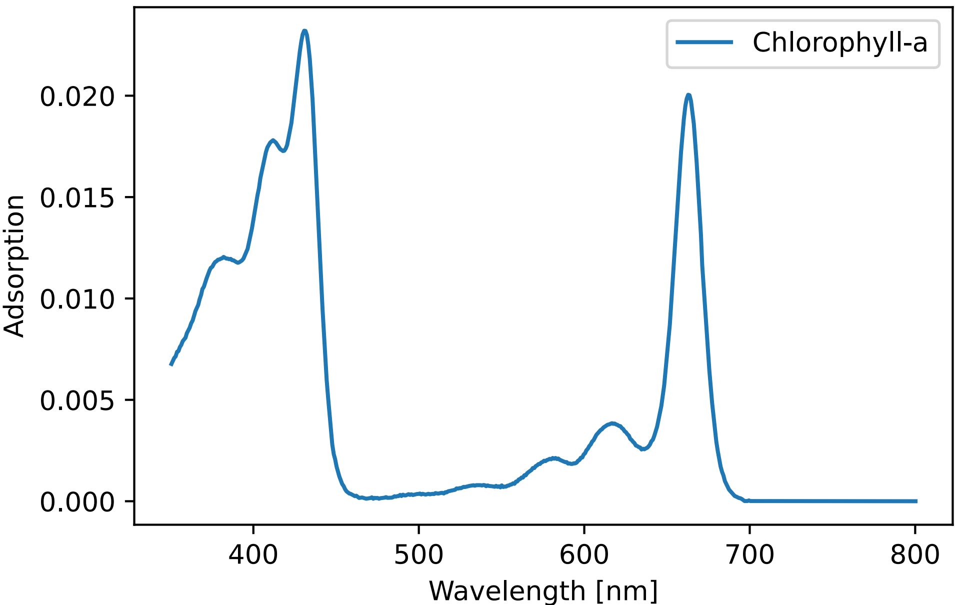
```
# This tell Python to display the plot.
plt.show()
```

Line 8
Here we plot by picking the x-axis as the wavelength from the dataset and the y-axis as measurements of chlorophyll-a

Line 11
Setting the x-axis name.

Line 14
Show the label in a little box, helps if we plot multiple curves in the same figure.

Line 17
Show the plot.



9.1.1 Exercise: Plotting

Use the cell below to plot some of the other columns of the dataset.
Try to plot several columns at the same time.

```
fig, ax = plt.subplots()

# Your code for plotting here

plt.show()
```

9.1.2 Exercise: Interpreting plots

Chlorophyll a is the main photosynthetic pigment, while chlorophyll b is an accessory pigment found in green algae and land plants. Use the plotted spectra to explain why having both pigments could be useful.

10. TØ 7. Extra

In this note we will work on the same dataset as the main note in which we saw how to plot data. Now we will look at how to manipulate the data, starting by converting the units of one column of the dataset and then doing a calculation that uses two columns.

Below is an excerpt of the dataset.

```
import pandas as pd
pigments = pd.read_csv('../../datasets/curated_abs_dataset.csv')
display(pigments)
```

	wavelength_nm	chlorophyll-a	chlorophyll-b	fucoxanthin	prasinoxanthin
0	800.2	0.0	0.000035	0.000629	0.000128
1	799.3	0.0	0.000000	0.000588	0.000166
2	798.5	0.0	0.000000	0.000599	0.000165
3	797.6	0.0	0.000000	0.000548	0.000216
4	796.7	0.0	0.000086	0.000491	0.000086
…	…	…	…	…	…
558	324.0	NaN	NaN	0.003124	0.002171
559	323.1	NaN	NaN	0.003056	0.002062
560	322.3	NaN	NaN	0.003098	0.002207
561	321.4	NaN	NaN	0.003092	0.002202
562	320.6	NaN	NaN	0.003042	0.002189

10.1 Data manipulations

We often need to transform parts of our data - e.g.

- Converting units
- Calculating derived properties
- Normalizing

Our dataset has the wavelength in nanometers, perhaps we would like to work with meters instead

```
pigments['wavelength_m'] = pigments['wavelength_nm'] * 10**(-9)
```

This is an example of an **element-wise operation** - every element of `pigments['wavelength_nm']` is multiplied by `10**(-9)`.

10.1.1 Exercise: Converting to photon energy

A more interesting unit conversion for this dataset would be to work with photon energy in eV, which can be computed with the formula

$$E = \frac{1240}{\lambda} \tag{10.1}$$

Where E is the photon energy and λ is the wavelength in eV.

In the cell below the dataset is available in the variable `pigments`, Use the cell to make this conversion to eV.

```
# Replace the blank line below with your code to make the conversion
_______

# Make a plot with the x-axis as photon energy
fig, ax = plt.subplots()


_______

plt.show()
```

10.1.2 Exercise: Mixture of pigments

We can approximate the absorbance of a mixture by using the equation

$$A_{\text{mixture}}(\lambda) = c_1 \cdot A_1(\lambda) + c_2 \cdot A_2(\lambda) \quad (10.2)$$

Where A are absorbances of each component of the mixture and c_i are proportions of each component. Use this to calculate the absorbance of a mixture of 70% `chlorophyll-a` and 30% `chlorophyll-b`

```
# Calculate the mixture
c1 = ______
c2 = ______
A_mixture = _______

# Plot the mixture absorbance
fig, ax = plt.subplots()

x_data = pigments['wavelength_nm']
ax.plot(x_data, pigments['chlorophyll-a'], label='c-a')
ax.plot(x_data, pigments['chlorophyll-b'], label='c-b')
______

plt.show()
```

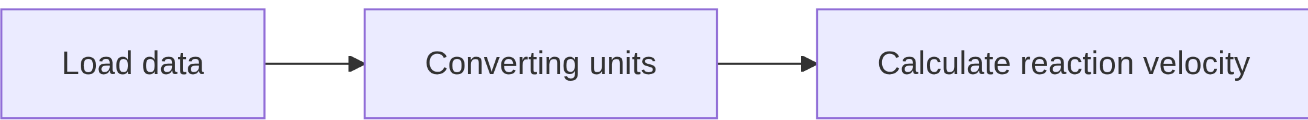
11. TØ 9. Laboratory Exercise 2

In this exercise you will analyze the data you've recorded in the laboratory. You will not be writing any Python code yourself - but you will walk through each of the step of analyzing this dataset.

In the "Fysiskbiokemi & Datanalayse"-course you will learn more about the theory behind this analysis and about the details of performing the analysis with Python.

11.1 Enzyme Kinetics

The diagram below describes the workflow we will be performing



11.1.1 Importing the data

The button below allows you to import your LØ dataset by uploading the `.xlsx`-file.

```
//| echo: false
viewof xlsx_file = Inputs.file({
  label: "Upload Excel file",
  accept: ".xlsx",
  required: false
})

xlsx_name = xlsx_file ? xlsx_file.name : null
xlsx_bytes = xlsx_file
  ? Array.from(new Uint8Array(await xlsx_file.arrayBuffer()))
  : null
```

! Question

What is shown in the different columns of the dataset?

11.1.2 Converting units

The measured absorbances are unitless, they are the log of the ratio between the incoming light intensity and the transmitted light intensity. We would like to work with concentrations instead, so we need to convert the absorbance to concentration. The conversion is done using Lambert-Beer's law,

$$A = \epsilon \cdot l \cdot c \Rightarrow c = \frac{\epsilon \cdot l}{A} \tag{11.1}$$

Where A is the measured absorbance, l is the path length, ϵ is the extinction coefficient and c is the concentration.

The cell below converts the absorbances to concentrations and creates a new dataframe.

Interactive Cell

```
df_conc = convert_absorbance(df)
display(df_conc)
```

Code

```
def lambert_beers(absorbance):
    L = 1 # cm
    ext_coeff = 18000 # 1 / (M cm)
    conc = absorbance / (L * ext_coeff) # M
    return conc


def convert_absorbance(df):
    data = {
        'time': df['time'],
    }

    for key in df.keys():
        if key == "time":
            continue

        new_key = f"S_{key}_mM"
        conc = lambert_beers(df[key])
        data[new_key] = conc

    df_conc = pd.DataFrame(data)

    return df_conc
```

The next cell plots the concentration as a function of time for each column, corresponding to a plot for each substrate concentration.

Interactive Cell

```
plot_dataframe(df_conc)
```

Code

```
def plot_dataframe(df):
    fig, ax = plt.subplots()

    for key in df.keys():
        if key == "time":
            continue

        ax.plot(df['time'], df[key], '-o')

    ax.set_ylabel('Concentration [M]')
    ax.set_xlabel('Time [s]')

    ax.patch.set_alpha(0.0)
    fig.patch.set_alpha(0.0)
    plt.show()
```

> ! Question
>
> What happens to the slope of the curves as substrate concentration increases?

11.1.3 Calculate reaction velocity

Hopefully what you see in the plot of concentrations vs. time are approximately linear relationships, as the next step is to find the slope of each of these - as that describes the initial reaction velocity. Mathematically we fit a linear function to each dataset

$$C = V_0 \cdot t + C_0 \tag{11.2}$$

Where V_0 is the intial reaction velocity that we are looking for.

Interactive Cell

```
# Uses fitting to the find the slope (reaction velocity)
# for each substrate concentration
substrate_conc, slopes = find_slopes(df_conc)

# Plots the substrate concentration vs. slope
fig, ax = plt.subplots()
ax.plot(substrate_conc, slopes, 'o')
ax.set_xlabel('Substrate concentration [M]')
ax.set_ylabel('Slope [M/s]')
plt.show()
```

Code

```
def linear_function(x, a, b):
    result = a * x + b
    return result

def find_slope(time_data, abs_data):
    ## This selects the times and absorbances that have been measured (not NaN)
    indices = np.where(~abs_data.isna())[0]
    time_data = time_data[indices]
    abs_data = abs_data[indices]

    ## Find the slope and the intercept using curve_fit and return the slope
    fitted_parameters, trash = curve_fit(linear_function, time_data, abs_data)
    slope = fitted_parameters[0]
    return slope

def find_slopes(df_conc):
    slopes = []
    substrate_concentrations = []

    for key in df_conc.keys():
        if key == "time":
            continue

        slope = find_slope(df_conc['time'], df_conc[key])
        conc = float(key.split('_')[1]) * 10**(-3)
        slopes.append(slope)
        substrate_concentrations.append(conc)

    substrate_concentrations = np.array(substrate_concentrations)
    slopes = np.array(slopes)
    return substrate_concentrations, slopes
```

11.1.4 Michaelis-Menten

As you will learn much more about in the "Fysiskbiokemi & datananalyse"-course this type of data can be described by a Michealis-Menten model

$$V_0 = \frac{V_{max} \cdot S}{K_M + S} \tag{11.3}$$

where V_0 is the initial reaction velocity, V_{max} is maximum reaction velocity occuring at high enough substrate concentrations and S is the substrate concentration.

Interactive Cell

```
K_M, V_max = fit_michealis_menten(substrate_conc, slopes)

plot_michealis_menten(K_M, V_max, substrate_conc, slopes)
```

Code

```
def michaelis_menten(S, K_M, V_max):
    result = (V_max * S) / (K_M + S)
    return result

def fit_michealis_menten(substrate_conc, slopes):
    K_M_estimate = 0.00001
    V_max_estimate = 8 * 10**(-8)
    initial_guess = [K_M_estimate, V_max_estimate]
     fitted_parameters, trash = curve_fit(michaelis_menten, substrate_conc, slopes,
initial_guess)
    K_M_fit, V_max_fit = fitted_parameters

    return K_M_fit, V_max_fit

def plot_michealis_menten(km, vmax, substrate, slopes):
    fig, ax = plt.subplots()

    S_smooth = np.linspace(substrate.min(), substrate.max())
    V0_fit = michaelis_menten(S_smooth, km, vmax)

    ax.plot(substrate, slopes, 'o', label='Data')
    ax.plot(S_smooth, V0_fit, label='Fit')

    ## These add x and y axis labels.
    ax.set_xlabel('Substrate concentration (M)')
    ax.set_ylabel('$V_0$ (M/s)')

       ax.text(0.7,  0.5,  f'$K_M=${km:0.4}  M',  transform=ax.transAxes,  ha='right',
va='center', fontsize=14)
        ax.text(0.7,  0.4,  f'$V_{{max}}=${vmax:0.4}  M/s',  transform=ax.transAxes,
ha='right', va='center', fontsize=14)

    plt.show()
```

! Question

What is an explanantion of this behaviour based on enzymekinetics?