Python and Programming 2

Programming workflow and advanced features

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6BER03), 2024-2025

Today's outline

- Scientific computing
  - Introduction
  - Introduction to NumPy
  - Math with NumPy
  - Array operations
- Plotting with Matplotlib
  - Line plots
  - Different plot styles
- IO
- Coding style
  - Program design
- Debugging and profiling
- Concluding remarks
- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- Numerically stable methods

Introduction to NumPy (1)

NumPy is a powerful library for numerical computing in Python. It introduces the multidimensional array (`ndarray`) data structure which is central to numerical computing tasks. To start using NumPy, you need to import it first:

```
>>> import numpy as np
```

Creating arrays from Python lists and accessing elements:

```
>>> arr = np.array([1, 2, 3, 4, 5])
>>> arr[0]
1
```

Create arrays with predetermined values:

```
>>> np.zeros(5) # Creates an array of five zeros
>>> np.ones((3, 3)) # Creates a 3x3 array of ones - note that the input argument is a tuple
```

Introduction to NumPy (2)

Useful array creation functions:

```
>>> arr = np.arange(0, 10, 2) # Creates an array with values from 0 to 10, step 2
>>> arr2 = np.linspace(0, 1, 5) # Creates an array with 5 values evenly spaced between 0 and 1
>>> arr3 = np.logspace(-3,2,6) # Creates an array spaced evenly on a logscale
```

Array operations:

```
>>> arr * 2 # Multiplies every element by 2
>>> arr + np.array([5, 4, 3, 2, 1]) # Element-wise addition
```

Inspecting your array:

```
>>> arr.shape # Returns the dimensions of the array (tuple)
>>> arr.shape[0] # Returns the number of rows ([1] for columns)
>>> len(arr) # Returns the size of the first dimension
>>> arr.size # Returns the total number of elements in arr
```

Introduction to NumPy (3)

Zeros- or ones-filled arrays:

```
>>> many_zeros = np.zeros_like(arr) # Array with zeros of size like arr, also: np.zeros
>>> just_ones = np.ones((2,4)) # Create array with ones of size (2,4), also: np.ones_like
>>> identity = np.eye((3,3)) # Identity matrix (zeros with ones on the main diagonal)
```

Or random numbers, useful for e.g. stochastic simulations. Note that these are NumPy intrinsic methods, and differ from the ones in the `random` module.

```
>>> np.random.rand() # Single random floating point number [0,1)
0.11879317099184983

>>> np.random.rand(2,3)
array([[0.36482694, 0.47204049, 0.20908193],
       [0.76626339, 0.64075302, 0.82440279]])

>>> np.random.randint(1,10,5) # 5 random integers from 1 (inclusive) until 10 (exclusive)
array([9, 1, 6, 4, 6])
```

Math with NumPy (1)

NumPy provides a rich set of functions to perform mathematical operations on arrays. Let's explore some categories of these operations.

- Linear Algebra:

```
>>> np.dot(arr, arr) # Dot product (alternative: arr @ arr)
>>> np.linalg.norm(arr) # Euclidean norm (L2 norm)
```

- Trigonometric Functions:

```
>>> np.sin(arr) # Sine of each element
>>> np.cos(arr) # Cosine of each element
```

- Statistical functions to analyze data:

```
>>> np.mean(arr) # Mean value of the array elements
>>> np.std(arr) # Standard deviation of the array elements
```

Math with NumPy (2)

Table: Useful NumPy Functions for Beginners

Function	Description
`np.array`	Create a NumPy array
`np.zeros`	Create an array filled with zeros
`np.ones`	Create an array filled with ones
`np.arange`	Create an array with evenly spaced values
`np.linspace`	Create an array with a specified number of evenly spaced values
`np.sin`	Sine function
`np.cos`	Cosine function
`np.tan`	Tangent function
`np.exp`	Exponential function
`np.log`	Natural logarithm
`np.sqrt`	Square root function
`np.dot`	Dot product of two arrays
`np.linalg.norm`	Calculate the norm of an array
`np.mean`	Calculate the mean of an array
`np.std`	Calculate the standard deviation of an array

Array operations (1)

In Python, NumPy ndarrays enable efficient operations on arrays, particularly mathematical operations associated with linear algebra. Let's look at how we can create and manipulate matrices using NumPy ndarrays:

- Manually creating matrices (2D ndarrays):

```
>>> import numpy as np
>>> matrix = np.array([[1, 2],
                       [3, 4]]) # Matrices are arrays of arrays(rows)
```

- Reshaping, slicing and modifying matrices (try for yourself and print each new result):

```
>>> mat = np.arange(25).reshape(5,5)
>>> col = mat[:,2] # Get column index 2
>>> row = mat[3,:] # Get row index 3
>>> sub = mat[1:4,1:4] # Save submatrix
>>> mat[1,:] = row # Replace row index 1 by 'row'
>>> mat[mat>5] = -1 # Conditional slicing/update of values
```

Array operations (2)

The real power of NumPy lies in the vectorized computations:

- Omit loops, write computations in natural language
- Shorter code
- Much, much faster

```
import numpy as np
import time

start = time.time()
x = np.linspace(0,2*np.pi,10000000)
y = np.exp(-x) * (2+np.sin(2*np.pi*x))
total_time = time.time() - start

print(f'{total_time = }')
```

```
total_time = 0.35589122772216797
```

Array operations (3)

If you do need to iterate over the elements:

- Think twice if you *really* have to
- If the answer is still yes, you can use `np.nditer` or `np.ndenumerate`:

```python
import numpy as np

# A 2D array
data = np.array([[11,12,13], [14,15,16]])

# Loop over each element using nditer
for val in np.nditer(data):
    print(f"Value: {val}")

# Loop using ndenumerate (gives index + value)
for i,val in np.ndenumerate(data):
    print(f"Enumerate index {i} has value {val};
          also {data[i]}")
```

```
Value: 11
Value: 12
Value: 13
Value: 14
Value: 15
Value: 16
Enumerate index (0, 0) has value 11; also 11
Enumerate index (0, 1) has value 12; also 12
Enumerate index (0, 2) has value 13; also 13
Enumerate index (1, 0) has value 14; also 14
Enumerate index (1, 1) has value 15; also 15
Enumerate index (1, 2) has value 16; also 16
```

Practice

Given a function $f(x) = x^2 + 2x - 4$

- Create a large `np.linspace` of $x \in [0, 20]$ e.g. with 1M numbers
- Compute $f(x)$ iteratively (i.e. with a .loop)
- Compute $f(x)$ vectorized (i.e. using NumPy operations)
- Compare timings

Assignment of arrays by reference

Careful with assignment, default is a reference to the same object as observed with lists:

```
>>> arr = np.arange(0,10,1)
>>> print(arr)
[0 1 2 3 4 5 6 7 8 9]
>>> new_arr = arr
>>> new_arr2 = arr.copy()
>>> new_arr[4] = 11
>>> print(new_arr)
[ 0 1 2 3 11 5 6 7 8 9]
>>> print(arr) # :mind_blown: we did not change this, did we?
[ 0 1 2 3 11 5 6 7 8 9]
```

Saving and loading matrices:

```
>>> np.save('my_matrix.npy', matrix) # Save the matrix to a file
>>> loaded_matrix = np.load('my_matrix.npy') # Load the matrix from a file
```

Today's outline

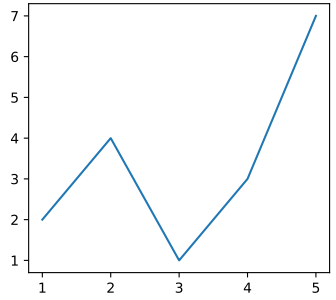
Introduction to Matplotlib

Matplotlib is a Python library used widely for creating static, animated, and interactive visualizations. To start, we first import the `pyplot` module from `matplotlib`.

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

Creating a simple line plot:

```
2 x = [1, 2, 3, 4, 5]
3 y = [2, 4, 1, 3, 7]
4 plt.plot(x, y)
5 plt.show()
```

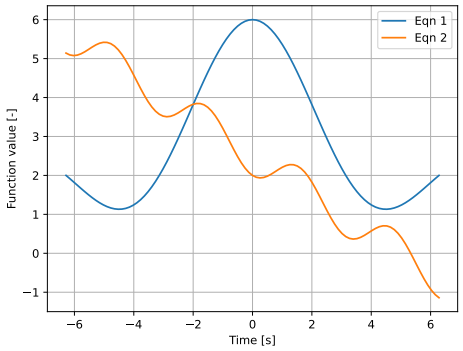


In Jupyter Notebooks, use `%matplotlib` to display figures in a separate window, or `%matplotlib inline` to display in the notebook.

Brushing up the graph

Adding multiple plots, axis labels and a legend:

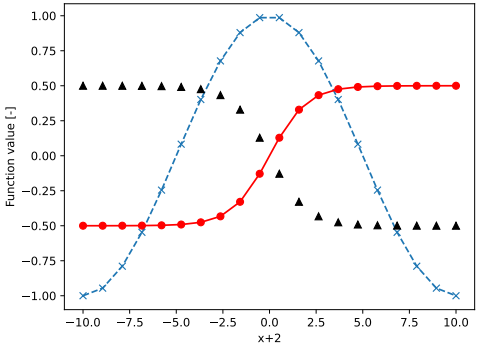
```
import matplotlib.pyplot as plt
import numpy as np

t = np.linspace(-2*np.pi, 2*np.pi,101)
y1 = 4*np.sin(t)/t + 2
y2 = np.sin(t)**2 - t/2 + 2

plt.plot(t,y1,label='Eqn 1')
plt.plot(t,y2,label='Eqn 2')
plt.xlabel('Time [s]')
plt.ylabel('Function value [-]')
plt.legend()
plt.grid()
plt.show()
```



Line plot styles

Changing markers, line styles:

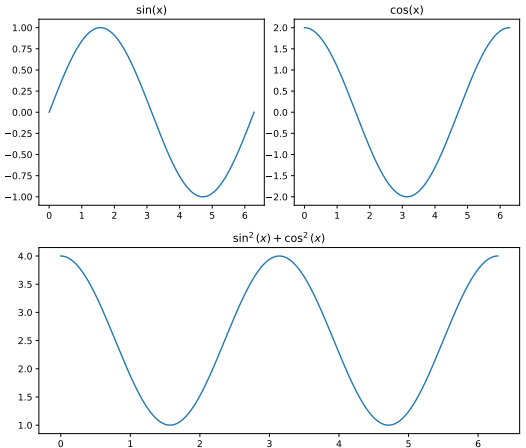
```
import matplotlib.pyplot as plt
import numpy as np

t = np.linspace(-10.0, 10.0, 20)
p = np.cos(np.pi * t/10)
q = 1/(1+np.exp(-t)) - 0.5

plt.plot(t,p,'--x') # Dashed line, cross
plt.plot(t,q,'r-o') # Red, line, circles
plt.plot(t,-q,'k^') # Black triangle marker
plt.xlabel('x+2')
plt.ylabel('Function value [-]')
plt.show()
```



Subplots

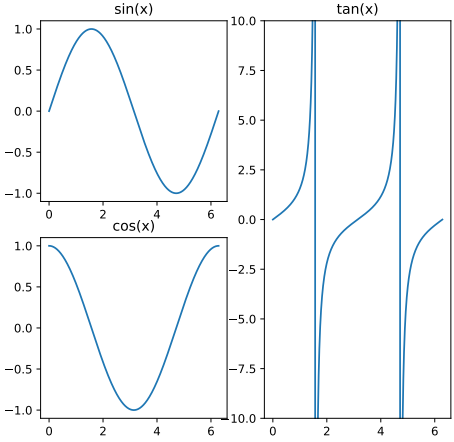
Multiple plots in a single figure are made by specifying a grid using subplot:

```
x = np.linspace(0,2*np.pi,1000)
y1,y2 = np.sin(x), 2*np.cos(x)

# Specify figure size
fig = plt.figure(figsize=(8, 7))

# First index of 2x2 grid (top-left)
ax1 = plt.subplot(2, 2, 1)
ax1.plot(x,y1)
ax1.set_title('sin(x)')

# Second index of 2x2 grid (top-right)
ax2 = plt.subplot(2, 2, 2)
ax2.plot(x,y2)
ax2.set_title('cos(x)')

# Second index of 2x1 grid (whole bottom)
ax3 = plt.subplot(2, 1, 2)
ax3.plot(x,y1**2+y2**2)
ax3.set_title(r'$\sin^2(x)+\cos^2(x)$')

plt.tight_layout()
plt.show()
```



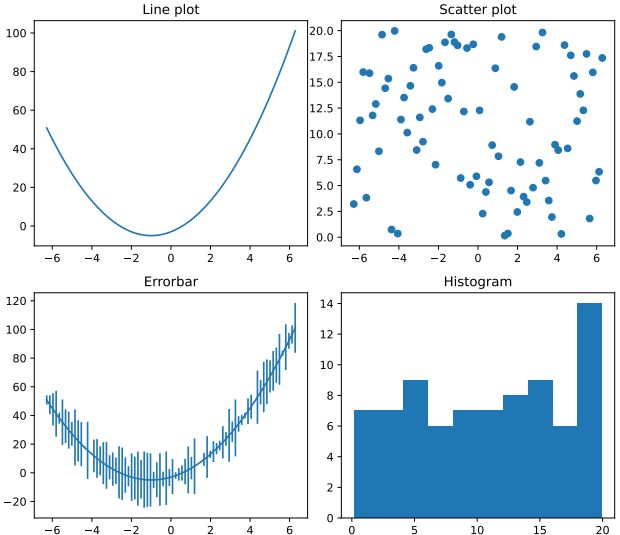
Practice

Try to create the following figure:



2D plot styles

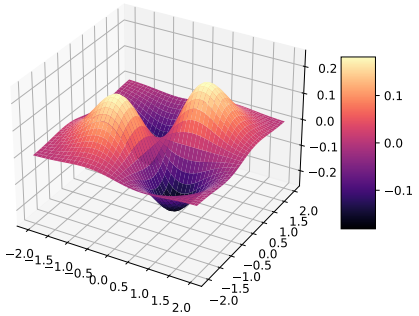
```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  x = np.linspace(-2*np.pi,2*np.pi,80)
5  y2 = 2*x**2 + 4*x - 3
6  y3 = 20*np.random.rand(y2.size)
7
8  ax1 = plt.subplot(2, 2, 1)
9  ax1.plot(x,y2)
10 ax1.set_title('Line plot')
11
12 ax2 = plt.subplot(2, 2, 2)
13 ax2.scatter(x,y3)
14 ax2.set_title('Scatter plot')
15
16 ax3 = plt.subplot(2, 2, 3)
17 ax3.errorbar(x,y2,yerr=y3)
18 ax3.set_title('Errorbar')
19
20 ax4 = plt.subplot(2,2,4)
21 ax4.hist(y3)
22 ax4.set_title('Histogram')
23
24 plt.show()
```



Surface plot

We draw a surface plot of $f(x,y) = xye^{(-x^2-y^2)}$:

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Make data.
x = np.arange(-2, 2, 0.025)
y = np.arange(-2, 2, 0.025)
x,y = np.meshgrid(x, y)
z = x * y * np.exp(-x**2 - y**2)

# Plot the surface.
surf = ax.plot_surface(x,y,z,
      cmap=cm.magma,linewidth=0, antialiased=False)

# Customize the z axis.
ax.set_zlim(-0.25, 0.25)

# Add a color bar which maps values to colors.
fig.colorbar(surf, shrink=0.5, aspect=5)

plt.show()
```



Advanced plotting: Animating plots

The `fig.canvas.draw()` and `fig.canvas.flush_events()` methods hold the execution of your program until the graph is updated, facilitating a live view of the simulation result.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 4*np.pi, 100)
y = np.sin(x)

fig, ax = plt.subplots()
line, = ax.plot(x, y, '-')

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_title('Animating a Sine Wave')
plt.show(block=False)

for i in range(len(x)):
    line.set_data(x[:i+1], y[:i+1])
    fig.canvas.draw()
    fig.canvas.flush_events()
    plt.pause(0.01)
```

Today's outline

Input and Output in Python (1)

Many programs require some input (data) to function correctly. A combination of the following is common:

- Input may be given in a parameters file (“hard-coded”)
- Input may be entered via the keyboard using the ‘input‘ function:

```
>>> a = input('Please enter a number: ')
```

- Input may be read from a file, for instance using Python’s built-in open function or libraries like ‘numpy‘ for more complex data structures:

```
with open('myData.txt', 'r') as file:
   data = file.read() # Alt: data.readlines() gives a list of lines

import numpy as np
data = np.loadtxt("my_file.csv")
```

- There are many other libraries and functions for more advanced input operations, such as json, xml, etc.

Input and Output in Python (2)

Output of results can be done in several ways, including:

- Displaying results to the console - simply omitting a print statement will automatically show expression results in most Python IDEs.
- Using the 'print' function to show results in the console:

```
>>> print("The result is:", result)
```

- Saving data to a file can be done using various methods including writing to a file or using libraries like pandas for structured data:

```
>>> with open('output.txt', 'w') as file:
... file.write(str(data))

>>> data.save("data.npy")
```

- More advanced output methods can utilize libraries such as NumPy, pandas, etc. to save data in various formats including JSON, Excel, etc.

Today's outline

Make a habit of the following adage

MAKE IT **WORK**

MAKE IT **RIGHT**

MAKE IT **FAST**

Make it work

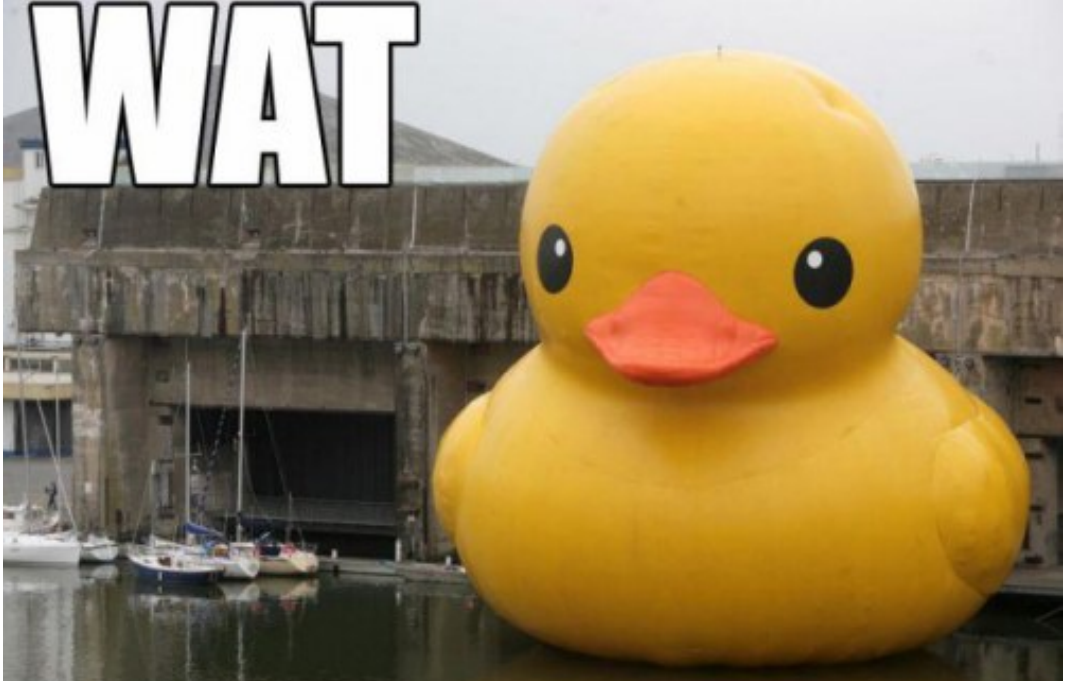
Use the building blocks of previous lecture to create an algorithm:

1. *Problem analysis*
   Contextual understanding of the nature of the problem to be solved
2. *Problem statement*
   Develop a detailed statement of the mathematical problem to be solved with the program
3. *Processing scheme*
   Define the inputs and outputs of the program
4. *Algorithm*
   A step-by-step procedure of all actions to be taken by the program (*pseudo-code*)
5. *Program the algorithm*
   Convert the algorithm into a computer language, and debug until it runs
6. *Evaluation*
   Test all of the options and conduct a validation study

Now it's time to make it right!

Interpret the following code

```
s = checksc()
if s:
    a = cb()
    b = cfrsp()
    if a < 5:
        if b > 5:
            a = gtbs()
        if a > b:
            ubx()
else:
    brn()
    gtbs()
```


WAT

Enhancing Readability with Proper Indentation

Proper indentation is not just about syntax in Python; it significantly impacts the readability of the code. Python recommends 4 spaces for indentation. Let's see the difference:

```
s = checksc()
if s:
a = cb()
b = cfrsp()
if a < 5:
if b > 5:
a = gtbs()
if a > b:
ubx()
else:
brn()
gtbs()
```

```
s = checksc()
if s:
    a = cb()
    b = cfrsp()
    if a < 5:
        if b > 5:
            a = gtbs()
        if a > b:
            ubx()
else:
    brn()
    gtbs()
```

Readable Variables and Function Names

Making use of descriptive variable and function names enhances the readability and understanding of the code.

```
s = checksc()
if s:
    a = cb()
    b = cfrsp()
    if a < 5:
        if b > 5:
            a = gtbs()
        if a > b:
            ubx()
else:
    brn()
    gtbs()
```

```
isAvailable = checkSchedule()
if isAvailable:
    bookCount = countBooks()
    freeShelfSpace = checkFreeShelfSpace()
    if bookCount < 5:
        if freeShelfSpace > 5:
            bookCount = visitBookStore()
        if bookCount > freeShelfSpace:
            useStorageBox()
else:
    burnBooks()
    visitBookStore()
```

Avoiding Magic Numbers in the Code

Magic numbers are constant values without a name, which can reduce code readability.
Replacing them with named constants can enhance the understanding of the code.

```
isAvailable = checkSchedule()
if isAvailable:
    bookCount = countBooks()
    freeShelfSpace = checkFreeShelfSpace()
    if bookCount < 5:
        if freeShelfSpace > 5:
            bookCount = visitBookStore()
        if bookCount > freeShelfSpace:
            useStorageBox()
else:
    burnBooks()
    visitBookStore()
```

```
MAX_SHELF_SPACE = 5
MIN_BOOKS_REQUIRED = 5

isAvailable = checkSchedule()
if isAvailable:
    bookCount = countBooks()
    freeShelfSpace = checkFreeShelfSpace()
    if bookCount < MAX_SHELF_SPACE:
        if freeShelfSpace > MIN_BOOKS_REQUIRED:
            bookCount = visitBookStore()
        if bookCount > freeShelfSpace:
            useStorageBox()
else:
    burnBooks()
    visitBookStore()
```

Now, That's More Like It!

Demonstrating the evolution of the script to a more readable and maintainable version.

```
s = checksc()
if s:
    a = cb()
    b = cfrsp()
    if a < 5:
        if b > 5:
            a = gtbs()
        if a > b:
            ubx()
else:
    brn()
    gtbs()
```

```
MAX_SHELF_SPACE = 5
MIN_BOOKS_REQUIRED = 5

isAvailable = checkSchedule()
if isAvailable:
    bookCount = countBooks()
    freeShelfSpace = checkFreeShelfSpace()
    if bookCount < MAX_SHELF_SPACE:
        if freeShelfSpace > MIN_BOOKS_REQUIRED:
            bookCount = visitBookStore()
        if bookCount > freeShelfSpace:
            useStorageBox()
else:
    burnBooks()
    visitBookStore()
```

Writing readable code

Good code reads like a book.

- When it doesn't, make sure to use comments. In Python, everything following `#` is a comment
- Prevent "smart constructions" in the code
- Re-use working code (i.e. create functions for well-defined tasks).
- Document functions using docstrings
- External documentation is also useful, but harder to maintain.

How not to comment

- Useless:

```
# Start program
```

- Obvious:

```
if (a > 5) # Check if a is greater than 5
    ...
```

- Too much about the life:

```
# Well... I do not know how to explain what is going on
# in the snippet below. I tried to code in the night
# with some booze and it worked then, but now I have a
# strong hangover and some parameters still need to be
# worked out...
```

- ...

```
# You may think that this function is obsolete, and doesn't seem to
# do anything. And you would be correct. But when we remove this
# function for some reason the whole program crashes and we can't
# figure out why, so here it will stay.
```

Adding comments to our Python program

Use comments to document design and purpose (functionality), not mechanics (implementation).

```
IAmFree = checkSchedule()
if IAmFree:
    # Count books and amount of free space on a shelf.
    # If minimum number of books I need is less than a
    # shelf capacity, go shopping and buy additional
    # literature. If the amount of books after the
    # shopping is too big, use boxes to store them.
    books = countBooks()
    shelfSize = countFreeSpaceShelf()

    ...

else:
    burnBooks()
    goToBookStore()
```

What else makes a good program?

- Portability (guaranteed in Python)
- Readability
- Efficiency
- Structural
- Flexibility
- Generality
- Documentation

Funny thing is: This list does not mention that the program should be actually working for its intended purposes!

Readability

Don’t use meaningless variable or function names. Rule of thumb: use verbs for functions and nouns for variables.

```
# stupid names
x = 5;
xx = myfunction(x);

# proper names
number_dams = 6;
beaver_workforce = allocate_beavers();
dams = build_dams(beaver_workforce, number_dams);
```

Efficiency

This one is difficult. Not much you can do without truly understanding how Python is utilizing your processor and memory. A couple of guidelines though:

- Avoid loops
- Especially avoid nested loops
- Use inherent matrix operations when possible
- Reduce IO (i.e. reading / writing to and from files)
- Don’t run scripts from network disks
- Pre-allocate your arrays, that means, making it as large as the maximum required size for your particular problem
- Use Python’s built-in modules for performance profiling, such as cProfile or timeit, to test the execution times

Efficiency: Measuring execution time example

```python
import numpy as np
import time

x = np.linspace(0, 10, 1000001)
start_time = time.time()

y = []
for cr in range(len(x)):
   y.append(np.sin(2 * np.pi * x[cr]))

end_time = time.time()
print(f'Execution time: {end_time - start_time}
     seconds')
```

```python
import numpy as np
import time

x = np.linspace(0, 10, 1000001)
start_time = time.time()

y = np.sin(2 * np.pi * x)

end_time = time.time()
print(f'Execution time: {end_time - start_time}
     seconds')
```

Structural

- Compartmentalize your code.
- Write functions whenever possible.
  - If you have > 15 lines of code, you can probably replace it by one or more functions.
  - In principle, it should not even matter how function works, as long as it gives the expected output.

- Put critical variables at the beginning of your program.

Structural

Write code as though it were paragraphs in a story.

```
from dependencies import * # importing all dependencies

# Step 0: Define variables
n_steps = 10000 # number of steps
n_walks = 1000 # number of random walk samples

# Step 1: Generate n_walks number of random walks
de = np.zeros((n_walks, 2))
for i in range(n_walks):
   angles = get_random_angles(n_steps)
   coord = transform_angles_to_coordinates(angles)
   de[i, 0] = calculate_de(coord) # store de
   de[i, 1] = de[i, 0]**2 # store de^2

# Step 2: Plot the histogram
plt.hist(de[:, 0], density=True)
D, P = calculate_pdf(n_steps, 1000)
plt.plot(D, P)
plt.show()
```

Flexibility

If you want to add a feature or change something inside the program, it should not require rewriting the whole program. (jargon: non-linear propagation of change).
Solution: Encapsulate your code and "Don't Repeat Yourself"

- Use functions for specific tasks (can you verbalize it? Then it is probably a function)
- Use variables, even for constants (it sounds like a oxymoron, but it's not! In fact, constant variables are a real thing)
- Use abstraction whenever possible

Generalization

If your code works for one problem, it should also work for a similar problem, whether it's in another company or on another planet.

Pro-tip: Separate data from algorithms.
To be honest, I rarely see students making this mistake.

```
# Stupid code
A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] # Hardcode data in the program

# Smart code
with open('some_random_dataset.dat') as file:
    data = file.read() # or use appropriate data loading functions from libraries like pandas
```

Documentation

Properly document your code. Write your comments in a clear and concise fashion. Help your future self: Write clear and concise documentation.

```
def f(a, b=None):
    if b is not None:
        c = a + b
    elif b is None and a is not None:
        c = a + a
    else:
        c = 0
    return c
```

Documentation

Properly document your code. Write your comments in a clear and concise fashion. Help your future self: Write clear and concise documentation.

```python
def f(a, b=None):
    """
    Add two values together.

    Parameters:
    a: The first number.
    b: The second number. Optional.

    Returns:
    The sum of a and b, or 2*a if b is not given, or 0 if a
          is not given.

    See also: sum, operator.add
    """
    if b is not None:
        c = a + b # sum two different numbers
    elif b is None and a is not None:
        c = a + a # double single number
    else:
        c = 0 # input not valid
    return c
```

Make a habit of the following adage

MAKE IT **WORK**

MAKE IT **RIGHT**

MAKE IT **FAST**

Make a habit of the following adage

1. *Make it work*
   Create an algorithm that does the intended job. Make sure it works, and works repeatedly. Test and verify frequently. Add *todo* comments when you're not sure about a certain decision.
2. *Make it right*
   Refactor the code to improve the code design. Insert functions, comments, compartmentalize it. Get rid of magic numbers, use sensible variable names. Check input. Test and verify. Align with the team!
3. *Make it fast*
   Measure and tune the performance of your code (profiling tool). In Python, vectorized calculations are much (!) faster than for-loops. Use sensible numerical techniques (e.g. higher-order integration).

Program by iterating over these aspects multiple times, starting at the fine-grained level, working your way up.

Today's outline

Errors in computer programs

The following symptoms can be distinguished:

- Unable to execute the program
- Program crashes, warnings or error messages
- Never-ending loops
- Wrong (unexpected) result

Three error categories:

Syntax errors	You did not obey the language rules. These errors prevent running or compilation of the program.
Runtime errors	Something goes wrong during the execution of the program resulting in an error message (problem with input, division by zero, loading of non-existent files, memory problems, etc.)
Semantic errors	The program does not do what you expect, but does what have told it to do.

Validation

- Testcases: run the program with parameters such that a known result is (should be) produced.
- Testcases: what happens when unforeseen input is encountered?
  - More or fewer arguments than anticipated? (Python uses `*args` and `**kwargs` to create a varying number of input arguments, and to check the number of given input arguments
  - Other data types than anticipated? How does the program handle this? Warnings, error messages (crash), NaN or worse: a program that silently continues?
- For physical modeling, we typically look for analytical solutions
  - Sometimes somewhat stylized cases
  - Possible solutions include Fourier-series
  - Experimental data

But: validation can only tell you *if* something is wrong, not *where* it went wrong.

The debugger (1)

- No-one can write a 1000-line code without making errors
  - If you can, please come work for us
- One of the most important skills you will acquire is debugging.
- Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.
- In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.
- Actually, you are the detective, the murderer and the victim at the same time.

"When you have eliminated the impossible, whatever remains, however improbable, must be the truth."

— A. Conan Doyle, The Sign of Four

The debugger (2)

A debugger can help you to:

- Pause a program at a certain line: set a *breakpoint*
- Check the values of variables during the program
- Controlled execution of the program:
  - One line at a time
  - Run until a certain line
  - Run until a certain condition is met (conditional breakpoint)
  - Run until the current function exits
- Note: You may end up in the source code of Python functions!
- Check Canvas (Python Crash Course section) for a demonstration of the debugger.

Recursive Fibonacci

- Create a program that computes the n-th Fibonacci number using recursion: $F_n = F_{n-1} + F_{n-2}$ with $F_1 = 1$ and $F_2 = 1$

```
1  def fibonacci_recursive(N):
2      """
3      Prints out the Nth Fibonacci number to the screen.
4      SYNTAX: fibonacci_recursive(N)
5      """
6      if N > 2:
7          Nminus1 = fibonacci_recursive(N-1)
8          Nminus2 = fibonacci_recursive(N-2)
9          out = Nminus1 + Nminus2
10     elif N == 1 or N == 2:
11         out = 1
12     else:
13         raise ValueError('Input argument was invalid')
14     return out
```

- Place a breakpoint line 5 (click on dash or press F12), run `fibonacci_recursive(5)`
- Explore the function of step F10, step into F11, and how the local workspace changes
- Stop the debugger (red stop button on top, or Shift+F5)
- Right-click the breakpoint, select *Set/modify condition*, enter `N==2`, run again.

Today's outline

Advanced concepts

- Object oriented programming: classes and objects
- Memory management: some programming languages require you to allocate computer memory yourself (e.g. for arrays)
- External libraries: in many cases, someone already built the general functionality you are looking for
- Compiling and scripting (“interpreted”); compiling means converting a program to computer-language before execution. Interpreted languages do this on the fly.
- Parallellization: Distributing expensive calculations over multiple processors or GPUs.

Make a habit of the following adage

MAKE IT **WORK**

MAKE IT **RIGHT**

MAKE IT **FAST**

Make a habit of the following adage

1. *Make it work*
   Create an algorithm that does the intended job. Make sure it works, and works repeatedly. Test and verify frequently. Add *todo* comments when you're not sure about a certain decision.
2. *Make it right*
   Refactor the code to improve the code design. Insert functions, comments, compartmentalize it. Get rid of magic numbers, use sensible variable names. Check input. Test and verify. Align with the team!
3. *Make it fast*
   Measure and tune the performance of your code (profiling tool). In Python, vectorized calculations are much (!) faster than for-loops. Use sensible numerical techniques (e.g. higher-order integration).

Program by iterating over these aspects multiple times, starting at the fine-grained level, working your way up.