Python and Programming 1

Programming basics and algorithms

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

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6BER03), 2024-2025

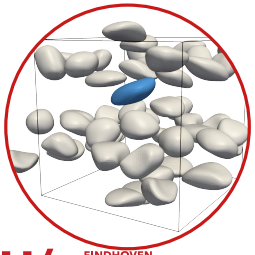
Today's outline

- Introduction
  - General programming
  - First steps
  - Further reading
- Data structures
  - Data types
  - Lists
  - Strings
  - Tuples
  - Dictionaries
- Control flow
  - Loops
  - Branching
- Functions
  - Defining functions
  - Recursion
  - Scope
  - Lambda functions
- Modules
  - Using modules
  - Math module
  - The random module
- Conclusions
- Exercises

Today's outline

Why should you learn something about programming?

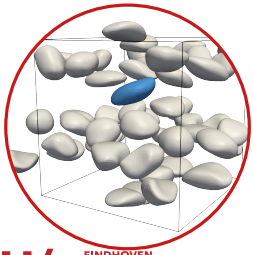
- Scientific analyses depend more than ever on computer programs and simulation methods

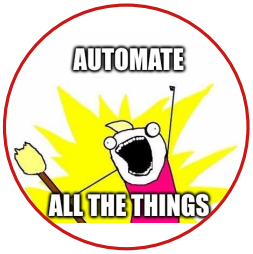


TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Why should you learn something about programming?

- Scientific analyses depend more than ever on computer programs and simulation methods
- Knowledge of programming allows you to automate routine tasks

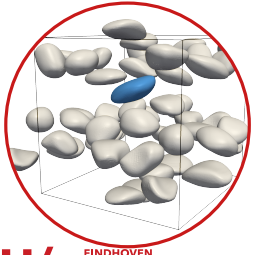


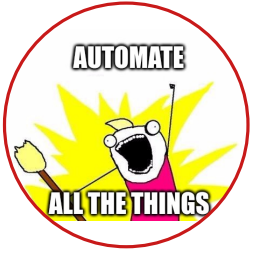


TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Why should you learn something about programming?

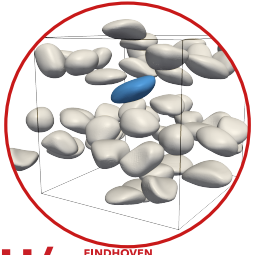
- Scientific analyses depend more than ever on computer programs and simulation methods
- Knowledge of programming allows you to automate routine tasks
- Ability to understand algorithms by inspection of the code

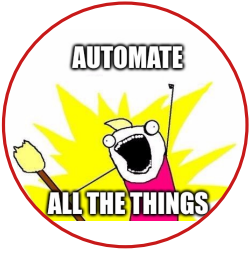




Why should you learn something about programming?

- Scientific analyses depend more than ever on computer programs and simulation methods
- Knowledge of programming allows you to automate routine tasks
- Ability to understand algorithms by inspection of the code
- Learn to think by dissecting a problem into smaller, easier to solve, parts





Introduction to programming

What is a program?

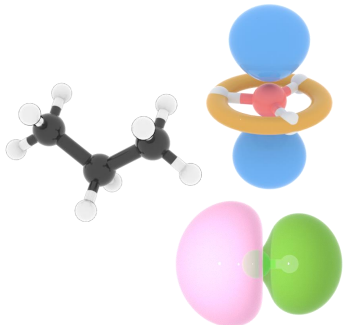
A program is a sequence of instructions that is written to perform a certain task on a computer.

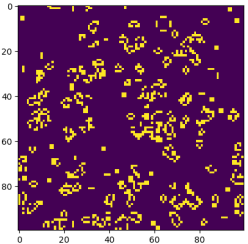
- The computation might be something mathematical, a symbolic operation, image analysis, etc.

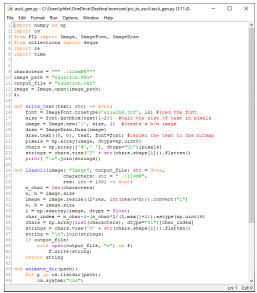
Program layout

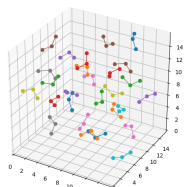
1. Input (Get the radius of a circle)
2. Operations (Compute and store the area of the circle)
3. Output (Print the area to the screen)

Versatility of Python



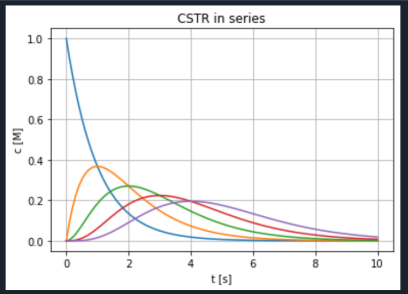






Versatility of Python: ODE solver

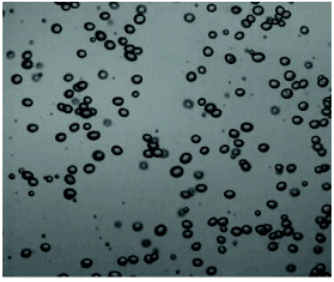
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

def ode(y, t):
    dydt = np.zeros(y.shape,float)
    dydt[0] = 0 - y[0]
    dydt[1:] = y[:-1] - y[1:]
    return dydt

t = np.linspace(0, 10, 100)
y0 = [1, 0, 0, 0, 0]
y = odeint(ode, y0, t)

plt.plot(t, y)
plt.grid()
plt.xlabel("t [s]")
plt.ylabel("c [M]")
plt.title("CSTR in series")
plt.show()
```




```
Python 3.10.9 | packaged by Anaconda, Inc. | (main, Mar  1 2023, 18:18:15) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 8.15.0 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/phfer/Downloads/temp.py', wdir='C:/Users/phfer/Downloads')

Important
Figures are displayed in the Plots pane by default. To make them also appear inline in the console, you need to
uncheck "Mute Inline plotting" under the options menu of Plots.

In [2]:
```

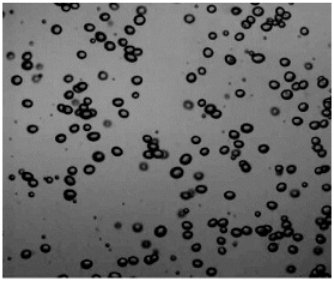
Versatility of Python: Image analysis

```python
# Importing necessary libraries
import numpy as np
from scipy import ndimage
from PIL.Image import fromarray
from skimage import io, color, feature, measure

# Loading and processing image
I = io.imread('bub0.png')
BW = color.rgb2gray(I)
E = feature.canny(BW)
F = ndimage.binary_fill_holes(E)

# Show final image
fromarray(F).show()
```



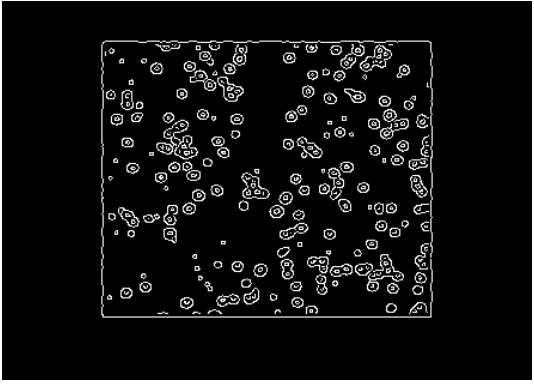
Versatility of Python: Image analysis

```python
# Importing necessary libraries
import numpy as np
from scipy import ndimage
from PIL.Image import fromarray
from skimage import io, color, feature, measure

# Loading and processing image
I = io.imread('bub0.png')
BW = color.rgb2gray(I)
E = feature.canny(BW)
F = ndimage.binary_fill_holes(E)

# Show final image
fromarray(F).show()
```



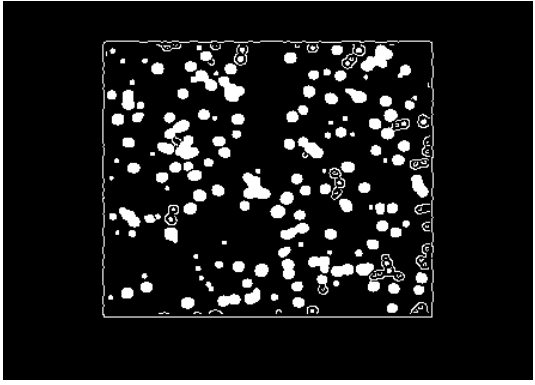
Versatility of Python: Image analysis

```
# Importing necessary libraries
import numpy as np
from scipy import ndimage
from PIL.Image import fromarray
from skimage import io, color, feature, measure

# Loading and processing image
I = io.imread('bub0.png')
BW = color.rgb2gray(I)
E = feature.canny(BW)
F = ndimage.binary_fill_holes(E)

# Show final image
fromarray(F).show()
```



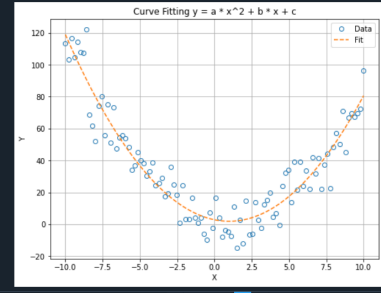
Versatility of Python: Image analysis

```python
# Importing necessary libraries
import numpy as np
from scipy import ndimage
from PIL.Image import fromarray
from skimage import io, color, feature, measure

# Loading and processing image
I = io.imread('bub0.png')
BW = color.rgb2gray(I)
E = feature.canny(BW)
F = ndimage.binary_fill_holes(E)

# Show final image
fromarray(F).show()
```



Versatility of Python: Curve fitting

```python
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

# Define the form of the function you want to fit
def func(x, a, b, c):
    return a * x**2 + b * x + c

# Generate example data
x = np.linspace(-10, 10, 100)
y = func(x, 1, -2, 3) + np.random.normal(scale=10, size=100)

# Use curve_fit to fit the function to the data
popt, _ = curve_fit(func, x, y)

# Generate y-data based on the fit
y_fit = func(x, *popt)

# Create a pretty plot
plt.figure(figsize=(8, 6))
plt.plot(x, y, 'o', mfc='none', label='Data')
plt.plot(x, y_fit, '--', label='Fit')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Curve Fitting y = a * x^2 + b * x + c')
plt.grid(True)
plt.legend()
plt.show()
```



Getting started

- Start the Python REPL (read–eval–print loop) by running `python` or `ipython`
- Enter the following commands on the command line. Evaluate the output.

Getting started

- Start the Python REPL (read–eval–print loop) by running `python` or `ipython`
- Enter the following commands on the command line. Evaluate the output.

```
>>> 2 + 3 # Some simple calculations
>>> 2 * 3
>>> 2 * 3**2 # Powers are done using **
```

Getting started

- Start the Python REPL (read–eval–print loop) by running `python` or `ipython`
- Enter the following commands on the command line. Evaluate the output.

```
>>> 2 + 3 # Some simple calculations
>>> 2 * 3
>>> 2 * 3**2 # Powers are done using **
>>> a = 2 # Storing values into the workspace
>>> b = 3
>>> c = (2 * 3)**2 # Parentheses set priority
>>> 10_000_000 / b
```

Getting started

- Start the Python REPL (read–eval–print loop) by running `python` or `ipython`
- Enter the following commands on the command line. Evaluate the output.

```
>>> 2 + 3 # Some simple calculations
>>> 2 * 3
>>> 2 * 3**2 # Powers are done using **
>>> a = 2 # Storing values into the workspace
>>> b = 3
>>> c = (2 * 3)**2 # Parentheses set priority
>>> 10_000_000 / b
>>> print(a)
>>> print(a,b)
>>> print(a,b,c,sep='--')
>>> print("Numerical methods")
```

Getting started

- Start the Python REPL (read–eval–print loop) by running `python` or `ipython`
- Enter the following commands on the command line. Evaluate the output.

```
>>> 2 + 3 # Some simple calculations
>>> 2 * 3
>>> 2 * 3**2 # Powers are done using **
>>> a = 2 # Storing values into the workspace
>>> b = 3
>>> c = (2 * 3)**2 # Parentheses set priority
>>> 10_000_000 / b
>>> print(a)
>>> print(a,b)
>>> print(a,b,c,sep='--')
>>> print("Numerical methods")
```

```
5
6
18



3333333.3333333335
2
2, 3
2--3--36
Numerical methods
```

Printing and formatting results

You can control the formatting of variables in string literals using various methods - we recommend f-strings. Note that formatting only changes how numbers are *displayed*, not the underlying representation.

```
>>> a = 19/4
>>> print("Few digits {:.2f}".format(a)) # 2 decimal places
>>> print("Many digits {:.10f}".format(a)) # 10 decimal places
>>>
>>> b = 22/7
>>> i = 13
>>> print("Almost pi: %1.4f" % b)
>>> print("i = %d, a = %1.4f and b = %1.8f" % (i,a,b))
>>>
>>> # Using f-strings (Python 3.6+)
>>> c = (21)**0.5 # sqrt of 21
>>> print(f"{c:.10f}") # Float with 10 decimal places
>>> mystr = f"{c:.2e}" # Scientific notation with 2 decimal places in a string object
>>> print(mystr) # Print the string object
>>> print(f"{b=}") # Use = to print variable name and value
>>> print(f"{b=:_^15.2}") # Adjust spacing and spacer character
```

A few helpful things

- Using the  and  keys, you can cycle through recent commands

A few helpful things

- Using the ↑ and ↓ keys, you can cycle through recent commands
- Typing part of a command and pressing Tab completes the command and lists the possibilities

A few helpful things

- Using the ↑ and ↓ keys, you can cycle through recent commands
- Typing part of a command and pressing Tab completes the command and lists the possibilities
- If a computation takes too long, you can press Ctrl+C to stop the program and return to the command line. Stored variables may contain incomplete results.

A few helpful things

- Using the ↑ and ↓ keys, you can cycle through recent commands
- Typing part of a command and pressing Tab completes the command and lists the possibilities
- If a computation takes too long, you can press Ctrl + C to stop the program and return to the command line. Stored variables may contain incomplete results.
- Sequences of commands (programs, scripts) are contained as py-files, plain text files with the `.py` extension.

A few helpful things

- Using the ↑ and ↓ keys, you can cycle through recent commands
- Typing part of a command and pressing Tab completes the command and lists the possibilities
- If a computation takes too long, you can press Ctrl+C to stop the program and return to the command line. Stored variables may contain incomplete results.
- Sequences of commands (programs, scripts) are contained as py-files, plain text files with the `.py` extension.
- Python scripts (and notes/markdown) can also be contained in jupyter notebooks, which have extension `.ipynb`.

A few helpful things

- Using the ↑ and ↓ keys, you can cycle through recent commands
- Typing part of a command and pressing Tab completes the command and lists the possibilities
- If a computation takes too long, you can press Ctrl+C to stop the program and return to the command line. Stored variables may contain incomplete results.
- Sequences of commands (programs, scripts) are contained as py-files, plain text files with the `.py` extension.
- Python scripts (and notes/markdown) can also be contained in jupyter notebooks, which have extension `.ipynb`.
- Such py-files must be in the *current working directory* or in the Python *path*, the locations where Python searches for a command. If you try to run a script that is not in the path, Python will throw an Exception/Error.

A few helpful things

- Using the ↑ and ↓ keys, you can cycle through recent commands
- Typing part of a command and pressing Tab completes the command and lists the possibilities
- If a computation takes too long, you can press Ctrl+C to stop the program and return to the command line. Stored variables may contain incomplete results.
- Sequences of commands (programs, scripts) are contained as py-files, plain text files with the `.py` extension.
- Python scripts (and notes/markdown) can also be contained in jupyter notebooks, which have extension `.ipynb`.
- Such py-files must be in the *current working directory* or in the Python *path*, the locations where Python searches for a command. If you try to run a script that is not in the path, Python will throw an Exception/Error.
- Anything following a `#` symbol is regarded as a comment

A few helpful things

- Using the ↑ and ↓ keys, you can cycle through recent commands
- Typing part of a command and pressing Tab completes the command and lists the possibilities
- If a computation takes too long, you can press Ctrl+C to stop the program and return to the command line. Stored variables may contain incomplete results.
- Sequences of commands (programs, scripts) are contained as py-files, plain text files with the `.py` extension.
- Python scripts (and notes/markdown) can also be contained in jupyter notebooks, which have extension `.ipynb`.
- Such py-files must be in the *current working directory* or in the Python *path*, the locations where Python searches for a command. If you try to run a script that is not in the path, Python will throw an Exception/Error.
- Anything following a `#` symbol is regarded as a comment
- There are several keyboard shortcuts (vary with text editor) that will make coding much more efficient.

Scripts, notebooks and REPL

- The REPL, indicated by the >>> prompt, has the advantage of immediate result after typing a command.
- Larger programs are better written in separate files; either Jupyter notebooks (.ipynb files) or plain script files (.py files).
- Defining functions in such files will put them in the *scope*, but will not run them until they are actually called.
- The snippets in these slides will continue to use the REPL for single-line commands, and move towards scripts when larger functions are being constructed.

Python help, documentation, resources

- Refer to the Python documentation at Official documentation.
  - Try for instance: `help(print)` or `help(help)`.
- Other packages that we will use:
  - NumPy documentation
  - Matplotlib documentation
  - SciPy documentation
- We supply a number of basic practice/reference modules: Python Crash Course.
- Python Crash Course, 3rd Edition by Eric Matthes
- A Whirlwind Tour of Python by Jake Vanderplas
- Introduction to Scientific Programming with Python by Joakim Sundnes
- Python Programming And Numerical Methods: A Guide For Engineers And Scientists by Kong, Siauw and Bayen
- Search the web, Reddit, YouTube, etc.

Today's outline

Terminology

Variable Piece of data stored in the computer memory, to be referenced and/or manipulated

Function Piece of code that performs a certain operation/sequence of operations on given input

Operators Mathematical operators (e.g. `+` `-` `*` or `/`), relational (e.g. `<` `>`or `==`, and logical operators (`and`, `or`)

Script Piece of code that performs a certain sequence of operations without specified input/output

Expression A command that combines variables, functions, operators and/or values to produce a result.

Variables in Python

- Python stores variables in the *namespace*
- You should recognize the difference between the *identifier* of a variable (its name, e.g. `x`, `setpoint_p`), and the data that it actually stores (e.g. 0.5)
- Python also defines a number of functions by default, e.g. **`min`**, **`max`** or **`sum`**.
  - A list of built-in methods is given by **`dir`**`(__builtins__)`
- You can assign a variable by the = sign:

```
>>> x = 4*3
>>> x
12
```

- If you don't assign a variable, it will be stored in _
- In most text editors, all variables are cleared automatically before the next execution.

Datatypes and variables

Python uses different types of variables:

Datatype	Example
`str`	`'Wednesday'`
`int`	`15`
`float`	`0.15`
`list`	`[0.0, 0.1, 0.2, 'Hello', ['Another','List']]`
`dict`	`{'name': 'word', "n": 2}`
`bool`	`False`
`tuple`	`(True, False)`

Datatypes and variables

Python uses different types of variables:

Datatype	Example
`str`	`'Wednesday'`
`int`	`15`
`float`	`0.15`
`list`	`[0.0, 0.1, 0.2, 'Hello', ['Another','List']]`
`dict`	`{'name': 'word', "n": 2}`
`bool`	`False`
`tuple`	`(True, False)`

Everything in Python is an object. You can use the `dir()` function to query the possible methods on an object of a datatype (e.g. `(dir(list))`, `dir(28)` or `dir("Yes!")`).

Lists in Python (1)

- Lists are containers of collections of objects
- A list is initialized using square brackets with comma-separated elements

```
>>> brands = ['Audi', 'Toyota', 'Honda', 'Ford', 'Tesla']
```

- Lists can contain and mix any object type, even other lists:

```
>>> another_list = [0.0, 0.1, 0.2, 'Hello', brands]
>>> print(another_list)
```

```
[0.0, 0.1, 0.2, 'Hello', ['Audi', 'Toyota', 'Honda', 'Ford', 'Tesla']]
```

- Access (i.e., read) an entry in a list. Note that indexing starts at 0:

```
>>> print(another_list[0],another_list[3])
```

```
0.0 Hello
```

Lists in Python (2)

- Manipulate the value of an entry goes likewise:

```
>>> another_list[3] = 'Bye' # Becomes: [0.0, 0.1, 0.2, 'Bye', ['Audi', ...]]
```

- Slicing is used to retrieve multiple elements:

```
>>> another_list[1:4] # This will give the elements from index 1 to index 3
```

```
[0.1, 0.2, 'Bye']
```

- Lists can be unpacked into individual variables:

```
>>> a,b,c,d,e = brands
>>> print(f"The first list element was {a}, then {b}, {c}, {d} and finally {e}.")
```

```
The first list element was Audi, then Toyota, Honda, Ford and finally Tesla.
```

- From here onwards, we will omit the `print` statements from the slides

Lists in Python (3)

- Lists can be concatenated or repeated by the addition and multiplication operators respectively:

```
>>> more_brands = ['Nissan','Kia'] + brands
```

```
['Nissan', 'Kia', 'Audi', 'Toyota', 'Honda', 'Ford', 'Tesla']
```

```
>>> zeros = 10*[0]
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- Find out which methods can be performed on a list by using `dir(more_brands)`:

```
more_brands.append('Volvo') # Append object (here: string literal) at the end of the list
more_brands.insert(1,'BMW') # Insert object at index 1
more_brands.sort() # Sorts the list in-place
item = more_brands.pop(3) # Removes element at index 3 from the list, stores it as item
```

Lists in Python (4)

Ranges of numbers are set using the `range(start=0,stop,step=1)` command:

- Create a list with a range of numbers:

```
>>> a = list(range(1, 11)) # Creates a list from 1 to 10
```

- List comprehensions can be used to create lists with more complex patterns:

```
>>> x = [i/10 for i in range(-10, 11)] # Creates a list from -1 to 1 with a step of 0.1
```

- Manipulating multiple components using slicing and a loop:

```
>>> y = list(range(11)) # Creates a list from 0 to 10
>>> for i in [0, 3, 4, 5, 6]:
>>> y[i] = 1
```

- Or (by supplying a list instead of a scalar):

```
>>> y[0:2] = [16, 19] # Sets y[0] to 16 and y[1] to 19
```

Assignment of variables; value or reference

Consider the following code snippets:

```
>>> i = 3
>>> j = i
>>> j = i + 3
>>> print(i,j)
>>> print(id(i), id(j)) # Print memory
      address of data
```

Assignment of variables; value or reference

Consider the following code snippets:

```
>>> i = 3
>>> j = i
>>> j = i + 3
>>> print(i,j)
>>> print(id(i), id(j)) # Print memory
      address of data
```

```
3, 6
8885416 8885544
```

Assignment of variables; value or reference

Consider the following code snippets:

```
>>> i = 3
>>> j = i
>>> j = i + 3
>>> print(i,j)
>>> print(id(i), id(j)) # Print memory
      address of data
```

```
3, 6
8885416 8885544
```

```
>>> list_a = ['aa', 1, 'bb', 12, True, 1.618]
>>> list_b = list_a
>>> list_b[2] = 'cc'
>>> print(list_a, list_b, sep='\n')
>>> print(id(list_a), id(list_b))
```

Assignment of variables; value or reference

Consider the following code snippets:

```
>>> i = 3
>>> j = i
>>> j = i + 3
>>> print(i,j)
>>> print(id(i), id(j)) # Print memory
      address of data
```

```
3, 6
8885416 8885544
```

```
>>> list_a = ['aa', 1, 'bb', 12, True, 1.618]
>>> list_b = list_a
>>> list_b[2] = 'cc'
>>> print(list_a, list_b, sep='\n')
>>> print(id(list_a), id(list_b))
```

```
['aa', 1, 'cc', 12, True, 1.618]
['aa', 1, 'cc', 12, True, 1.618]
140285003056512 140285003056512
```

Assignment of variables; value or reference

Consider the following code snippets:

```
>>> i = 3
>>> j = i
>>> j = i + 3
>>> print(i,j)
>>> print(id(i), id(j)) # Print memory
      address of data
```

```
3, 6
8885416 8885544
```

```
>>> list_a = ['aa', 1, 'bb', 12, True, 1.618]
>>> list_b = list_a
>>> list_b[2] = 'cc'
>>> print(list_a, list_b, sep='\n')
>>> print(id(list_a), id(list_b))
```

```
['aa', 1, 'cc', 12, True, 1.618]
['aa', 1, 'cc', 12, True, 1.618]
140285003056512 140285003056512
```

- Primitive or immutable data types (e.g. `int`, `float`, `str`, `tuple`) are *assigned by value*; the value is copied and changes do not affect the original variables.
- Mutable data types (e.g. `list`, `set`, `dict`) are *assigned by reference*; they are two names pointing to the same data, changing one affects the values of the other.

Practice

Given a vector

$$x = [2\ 4\ 6\ 8\ 10\ 12\ 14\ 16\ 18\ 20\ 30\ 40\ 50\ 60\ 70\ 80]$$

- Define the vector using `range`'s, without typing all individual elements

Practice

Given a vector

$$x = [2\ 4\ 6\ 8\ 10\ 12\ 14\ 16\ 18\ 20\ 30\ 40\ 50\ 60\ 70\ 80]$$

- Define the vector using `range`'s, without typing all individual elements

```
>>> x = list(range(2,20,2)) + list(range(20,90,10))
```

Practice

Given a vector

$$x = [2\ 4\ 6\ 8\ 10\ 12\ 14\ 16\ 18\ 20\ 30\ 40\ 50\ 60\ 70\ 80]$$

- Define the vector using `range`'s, without typing all individual elements

```
>>> x = list(range(2,20,2)) + list(range(20,90,10))
```

- Investigate the meaning of the following commands:

```
>>> x[2]
>>> x[0:5]
>>> x[:-1]
>>> y = x[4:]
>>> y[3]
>>> y.pop(3)
>>> sum(x)
>>> max(x)
>>> min(x)
>>> x[::-1]
```

Strings in Python (1)

Creating a string:

```
>>> s = "Hello, world!"
>>> len(s)
13
```

Accessing a character in a string:

```
>>> s[7]
'w'
```

Getting a substring:

```
>>> s[7:12]
'world'
```

Or separate by whitespace using a string method (see `dir(s)`):

```
>>> s.split()
['Hello,', 'world!']
```

Strings in Python (2)

Replacing a substring with another string:

```
>>> s.replace('world', 'Python')
'Hello, Python!'
```

Converting to upper and lower case:

```
>>> s.upper()
'HELLO, WORLD!'
>>> s.lower()
'hello, world!'
```

You can combine methods with string literals too:

```
>>> s.replace('WoRlD'.lower(), 'Python')
'Hello, Python!'
>>> s.startswith('hello'.title())
True
```

Finding the starting index of a substring:

```
>>> s.index("world")
7
```

Practice

Given a string

```
>>> s = "Python programming is fun!"
```

- Find and print the index of the word "is".
- Create a new string where "fun" is replaced with "awesome".
- Print the string in uppercase.

```
>>> s.index('is')
>>> p = s.replace('fun','awesome')
>>> print(p.upper())
```

Tuples in Python

A tuple is a built-in data type that contains an immutable sequence of values. Creating a tuple:

```
>>> t = (1, 2, 3)
```

Accessing an element of a tuple:

```
>>> t[1]
2
```

Tuples are immutable, so we can't change their elements. However, we can create a new tuple based on the old one:

```
>>> t = t + (4, )
```

Finding the length of a tuple:

```
>>> len(t)
4
```

Practice

Given a tuple

```
>>> t = (1, 2, 3, 4, 5, 6)
```

- Access and print the third element of the tuple.
- Try to change the value of the second element of the tuple.
- Create a new tuple by concatenating a second tuple $(7, 8, 9)$ to the original tuple.

Practice

Given a tuple

```
>>> t = (1, 2, 3, 4, 5, 6)
```

- Access and print the third element of the tuple.
- Try to change the value of the second element of the tuple.
- Create a new tuple by concatenating a second tuple $(7, 8, 9)$ to the original tuple.

```
t = (1, 2, 3, 4, 5, 6)
print(t[2])
t[2] = 6
t2 = t + (7,8,9)
print(t2)
```

Dictionaries in Python (1)

Creating a dictionary:

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
```

Accessing a value by its key:

```
>>> d['b']
2
```

Modifying a value associated with a key:

```
>>> d['b'] = 47
```

Adding a new key-value pair:

```
>>> d['d'] = 4
```

Removing a key-value pair using pop:

```
>>> d.pop('d')
4
```

Dictionaries in Python (2)

Get all keys as a list:

```
>>> list(d.keys())
['a', 'b', 'c']
```

Get all values as a list:

```
>>> list(d.values())
[1, 47, 3]
```

Get all key-value pairs as a list of tuples:

```
>>> list(d.items())
[('a', 1), ('b', 47), ('c', 3)]
```

Practice

Given a dictionary

```
>>> d = { 'Alice': 24, 'Bob': 27, 'Charlie': 22, 'Dave': 30}
```

- Access and print the age of 'Charlie'.
- Update 'Alice' age to 25.
- Add a new entry for 'Eve' with age 29.
- Print all the keys in the dictionary.

Practice

Given a dictionary

```
>>> d = { 'Alice': 24, 'Bob': 27, 'Charlie': 22, 'Dave': 30}
```

- Access and print the age of 'Charlie'.
- Update 'Alice' age to 25.
- Add a new entry for 'Eve' with age 29.
- Print all the keys in the dictionary.

```
1 >>> print(d['Charlie'])
2 >>> d['Alice'] = 25
3 >>> d['Eve'] = 29
4 >>> print(d.keys())
5 dict_keys(['Alice', 'Bob', 'Charlie', 'Dave', 'Eve'])
```

Today's outline

Loops in Python (1)

The `for` loop is used to iterate over a sequence (e.g. lists, sets, tuples, dictionaries, strings). Any *iterable* object can be listed over:

```
>>> for i in range(5):
... print(i)
0
1
2
3
4
```

Loops in Python (1)

The `for` loop is used to iterate over a sequence (e.g. lists, sets, tuples, dictionaries, strings). Any *iterable* object can be listed over:

```
>>> for i in range(5):
... print(i)
0
1
2
3
4
```

You can iterate over a list directly:

```
>>> my_list = [1, 2, 3, 4, 5]
>>> for num in my_list:
... print(num)
1
2
3
4
5
```

Loops in Python (2)

The `enumerate` keyword returns both the *index* as well as the *list element*:

```
>>> my_list = ['aa', 1, 'bb', 12, True, 1.618034, []]
>>> for idx,elm in enumerate(my_list):
... print(f'Element {elm} of type {type(elm)} at index {idx}')
```

Loops in Python (2)

The `enumerate` keyword returns both the *index* as well as the *list element*:

```
>>> my_list = ['aa', 1, 'bb', 12, True, 1.618034, []]
>>> for idx,elm in enumerate(my_list):
... print(f'Element {elm} of type {type(elm)} at index {idx}')
```

```
Element aa of type <class 'str'> at index 0
Element 1 of type <class 'int'> at index 1
Element bb of type <class 'str'> at index 2
Element 12 of type <class 'int'> at index 3
Element True of type <class 'bool'> at index 4
Element 1.618034 of type <class 'float'> at index 5
Element [] of type <class 'list'> at index 6
```

Loops in Python (3)

The ‘while‘ loop keeps going as long as a condition is `True`:

```
>>> i = 0
>>> while i < 3:
... print(i)
... i += 1
0
1
2
```

Loops in Python (3)

The ‘while‘ loop keeps going as long as a condition is `True`:

```
>>> i = 0
>>> while i < 3:
... print(i)
... i += 1
0
1
2
```

Use `break` to exit a loop prematurely, and `continue` to skip to the next iteration:

```
>>> for i in range(5):
... if i == 3:
... break
... print(i)
0
1
2
```

Practice

Given a list

```
>>> my_list = [1, 3, 7, 8, 9]
```

- Use a for loop to print each element of the list.

Practice

Given a list

```
>>> my_list = [1, 3, 7, 8, 9]
```

- Use a for loop to print each element of the list.
- Use a while loop to print the values at indices 0 to 4.

Practice

Given a list

```
>>> my_list = [1, 3, 7, 8, 9]
```

- Use a for loop to print each element of the list.
- Use a while loop to print the values at indices 0 to 4.
- Use a loop to find and print the index of the number 7 in the list.

```
>>> for i in my_list:
... print(i)
```

```
>>> c = 0
>>> while (c<4):
... print(f"{my_list[c]=}")
... c += 1
```

```
>>> c = 0
>>> while not my_list[c] == 7:
... c += 1
>>> print(my_list[c])
```

Conditional Statements in Python

The Boolean type `bool` has only 2 possible values: `True` or `False`
The `if` statement is used to execute a block of code only if a condition is evaluated to `True`:

```
>>> x = 5
>>> if x > 0:
... print("x is positive")
x is positive
```

Conditional Statements in Python

The Boolean type `bool` has only 2 possible values: `True` or `False`
The `if` statement is used to execute a block of code only if a condition is evaluated to `True`:

```
>>> x = 5
>>> if x > 0:
... print("x is positive")
x is positive
```

Use `elif` to specify additional conditions, and `else` to define what to do if no conditions are met:

```
>>> if x > 10:
... print("x is greater than 10")
... elif x == 10:
... print("x is exactly 10")
... else:
... print("x is less than 10")
x is less than 10
```

Nested conditionals

Nesting conditions allows for more complex conditionals:

```
>>> if x > 0:
... if x % 2 == 0: # The modulo operator % yields the remainder of a division
... print("x is positive and even")
... else:
... print("x is positive but odd")
... else:
... print("x is non-positive")
x is positive but odd
```

Nested conditionals

Nesting conditions allows for more complex conditionals:

```
>>> if x > 0:
... if x % 2 == 0: # The modulo operator % yields the remainder of a division
... print("x is positive and even")
... else:
... print("x is positive but odd")
... else:
... print("x is non-positive")
x is positive but odd
```

The `in` keyword can be used to check membership in a sequence:

```
>>> my_list = [1, 2, 3, 4, 5]
>>> if 3 in my_list:
... print("3 is a member of the list")
3 is a member of the list
```

Combined conditionals

Leap year determination

To be a leap year, the year number must be divisible by four, except for end-of-century years, which must be divisible by 400.

Combined conditionals

Leap year determination

To be a leap year, the year number must be divisible by four, except for end-of-century years, which must be divisible by 400.

We can create combined conditions using `not`, `and` and `or` to determine whether we have a leap year:

Combined conditionals

Leap year determination

To be a leap year, the year number must be divisible by four, except for end-of-century years, which must be divisible by 400.

We can create combined conditions using `not`, `and` and `or` to determine whether we have a leap year:

```
>>> year = 2024
>>> if year % 4 == 0 and (not year % 100 == 0 or year % 400 == 0):
... print(f"{year} is a leap year.")
... else:
... print(f"{year} is not a leap year.")
```

Conditionals in list comprehensions

List comprehensions can be extended with conditionals too:

```
>>> x = [i for i in range(0,31) if i%3 == 0]
>>> print(x)
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

Conditionals in list comprehensions

List comprehensions can be extended with conditionals too:

```
>>> x = [i for i in range(0,31) if i%3 == 0]
>>> print(x)
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

Conditions are not restricted to modulo's. Here we select artists who have an 'r' or 'R' in their name:

```
artists = ["Adele", "Harry Styles", "Stef Ekkel", "Ed Sheeran", "Nicki Minaj", "Ariana Grande", "
    Robbie Williams"]

my_artists = [artist for artist in artists if artist.lower().count('r') > 0]
print(my_artists)
['Harry Styles', 'Ed Sheeran', 'Ariana Grande', 'Robbie Williams']
```

Practice

Consider the list:

```
>>> my_list = [x**3 for x in range(1,25,2)] # Cubes of odd numbers
```

Practice

Consider the list:

```
>>> my_list = [x**3 for x in range(1,25,2)] # Cubes of odd numbers
```

- Use an `if` statement to check if 125 is in `my_list`, and print a message indicating the result.
- Write a statement that checks and prints whether `my_list[3]` is divisible by 3, and if not, print the remainder.
- Use a list comprehension to create a list of all numbers in `my_list` that are not divisible by 5.

Practice

Consider the list:

```
>>> my_list = [x**3 for x in range(1,25,2)] # Cubes of odd numbers
```

- Use an `if` statement to check if 125 is in `my_list`, and print a message indicating the result.
- Write a statement that checks and prints whether `my_list[3]` is divisible by 3, and if not, print the remainder.
- Use a list comprehension to create a list of all numbers in `my_list` that are not divisible by 5.

```
>>> if 125 in my_list:
... print('The number 125 was
      found in my_list!')
```

```
>>> if my_list[3] % 3 == 0:
... print(f'{my_list[3]} is
      divisible by 3')
... else:
... print(f'The remainder of {
      my_list[3]} and 3 is {
      my_list[3]%3}')
```

```
>>> k = [n for n in my_list if
      not n % 5 == 0 ]
```

Today's outline

Functions in Python (1)

Functions are defined using the `def` keyword followed by the function name and a list of parameters in parentheses. The function body starts after the colon:

```
>>> def greet(name):
...     print(f"Hello, {name}!")
```

Functions in Python (1)

Functions are defined using the `def` keyword followed by the function name and a list of parameters in parentheses. The function body starts after the colon:

```
>>> def greet(name):
...     print(f"Hello, {name}!")
```

Call the function with the necessary arguments:

```
>>> greet("Alice")
Hello, Alice!
```

Functions in Python (1)

Functions are defined using the **def** keyword followed by the function name and a list of parameters in parentheses. The function body starts after the colon:

```
>>> def greet(name):
... print(f"Hello, {name}!")
```

Call the function with the necessary arguments:

```
>>> greet("Alice")
Hello, Alice!
```

Functions can return values using the **return** keyword:

```
>>> def add(a, b):
... return a + b
```

Functions in Python (1)

Functions are defined using the `def` keyword followed by the function name and a list of parameters in parentheses. The function body starts after the colon:

```
>>> def greet(name):
... print(f"Hello, {name}!")
```

Call the function with the necessary arguments:

```
>>> greet("Alice")
Hello, Alice!
```

Functions can return values using the `return` keyword:

```
>>> def add(a, b):
... return a + b
```

Capture the return value in a variable, e.g. `result`:

```
>>> result = add(2, 3)
>>> print(result)
5
```

Functions in Python (2)

Default argument values can be specified, making the argument optional:

```
>>> def greet(name, greeting="Hello"):
... print(f"{greeting}, {name}!")
```

Functions in Python (2)

Default argument values can be specified, making the argument optional:

```
>>> def greet(name, greeting="Hello"):
... print(f"{greeting}, {name}!")
```

Call the function with or without the optional argument:

```
>>> greet("Bob")
Hello, Bob!
>>> greet("Bob", "Buzz off")
Buzz off, Bob!
```

Functions in Python (2)

Default argument values can be specified, making the argument optional:

```
>>> def greet(name, greeting="Hello"):
... print(f"{greeting}, {name}!")
```

Call the function with or without the optional argument:

```
>>> greet("Bob")
Hello, Bob!
>>> greet("Bob", "Buzz off")
Buzz off, Bob!
```

Python supports functions with a variable number of arguments:

```
>>> def my_function(*args):
... print(args)
```

Functions in Python (2)

Default argument values can be specified, making the argument optional:

```
>>> def greet(name, greeting="Hello"):
... print(f"{greeting}, {name}!")
```

Call the function with or without the optional argument:

```
>>> greet("Bob")
Hello, Bob!
>>> greet("Bob", "Buzz off")
Buzz off, Bob!
```

Python supports functions with a variable number of arguments:

```
>>> def my_function(*args):
... print(args)
```

Call the function with a varying number of arguments:

```
>>> my_function(1, 2, 3, "Hello")
(1, 2, 3, "Hello")
```

Functions in Python (3)

Functions can also return multiple values (also >2)

```
>>> def statistics(numbers):
... return max(numbers), min(numbers)
```

Functions in Python (3)

Functions can also return multiple values (also >2)

```
>>> def statistics(numbers):
... return max(numbers), min(numbers)
```

Let's call the function with some list:

```
>>> numlist = [94,12,6,19,33,14,81,56,43,22]
>>> print(statistics(numlist))
(94, 6)
```

Functions in Python (3)

Functions can also return multiple values (also >2)

```
>>> def statistics(numbers):
... return max(numbers), min(numbers)
```

Let's call the function with some list:

```
>>> numlist = [94,12,6,19,33,14,81,56,43,22]
>>> print(statistics(numlist))
(94, 6)
```

Store the elements in separate variables:

```
>>> a,b = statistics(numlist)
>>> print(f'{a=}, {b=}')
a=94, b=6
```

Functions in Python (4)

- Functions are very useful for *abstraction*:
  - You can compartmentalize and 'hide' complex pieces of code
  - Retain flexibility through arguments
  - You can reuse often used pieces of code, limiting copy-paste of code
- Extending functionality or fixing bugs is done in 1 place

Functions in Python (4)

- Functions are very useful for *abstraction*:
  - You can compartmentalize and 'hide' complex pieces of code
  - Retain flexibility through arguments
  - You can reuse often used pieces of code, limiting copy-paste of code
- Extending functionality or fixing bugs is done in 1 place
- **Documentation is crucial!**

Functions in Python (4)

- Functions are very useful for *abstraction*:
  - You can compartmentalize and 'hide' complex pieces of code
  - Retain flexibility through arguments
  - You can reuse often used pieces of code, limiting copy-paste of code
- Extending functionality or fixing bugs is done in 1 place
- **Documentation is crucial!**

```
>>> def statistics(numbers):
... """Return the maximum and minimum of a list of numbers
... Function arguments:
... numbers: list of numbers"""
... return max(numbers), min(numbers)
```

Functions in Python (4)

- Functions are very useful for *abstraction*:
  - You can compartmentalize and 'hide' complex pieces of code
  - Retain flexibility through arguments
  - You can reuse often used pieces of code, limiting copy-paste of code
- Extending functionality or fixing bugs is done in 1 place
- **Documentation is crucial!**

```
>>> def statistics(numbers):
... """Return the maximum and minimum of a list of numbers
... Function arguments:
... numbers: list of numbers"""
... return max(numbers), min(numbers)
```

```
>>> help(statistics)
```

```
Help on function statistics in module __main__:

statistics(numbers)
Return the maximum and minimum of a list of numbers
Function arguments:
numbers: list of numbers
```

Passing arguments by value or reference?

Recall that certain Python variables are assigned by value, and others reference; the same goes for passing arguments [1]:

- Primitive or immutable data types are *passed by value*; the value is copied and changes made inside the function will not affect the values stored in the variables passed to the function.
- Mutable data types (e.g. list, set, dict) are passed to a function *by reference*; changes that are made inside the function do affect the values outside of the function.

Consider the function:

```
1 def func(x, y):
2     x = x - 1 # Subtract 1
3     y.pop() # Remove last item
```

[1] https://k0nze.dev/posts/python-copy-reference-none/

Passing arguments by value or reference?

Recall that certain Python variables are assigned by value, and others reference; the same goes for passing arguments [1]:

- Primitive or immutable data types are *passed by value*; the value is copied and changes made inside the function will not affect the values stored in the variables passed to the function.
- Mutable data types (e.g. list, set, dict) are passed to a function *by reference*; changes that are made inside the function do affect the values outside of the function.

Consider the function:

```python
def func(x, y):
    x = x - 1 # Subtract 1
    y.pop() # Remove last item
```

```python
i = 1
l = ['a', 'b']

func(i, l)

print(i, l)
```

```
1, ['a']
```

[1] https://k0nze.dev/posts/python-copy-reference-none/

Practice

Define a function that computes the factorial of a number, $n!$:

$$n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$$

Compute $\exp(x)$ using the Taylor series, iterate until the change is smaller than $1 \cdot 10^{-6}$:

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}$$

Practice

Define a function that computes the factorial of a number, *n*!:

$$n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$$

```
def factorial(n):
    """Compute the factorial of n"""
    x = 1
    for i in range(1,n+1):
        x *= i
    return x
```

Compute $\exp(x)$ using the Taylor series, iterate until the change is smaller than $1 \cdot 10^{-6}$:

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!}$$

```
def exponential(x, eps=1.0e-6):
    """Compute exponential of x with accuracy eps
          (default: 1.0e-6)"""
    i = 0
    taylor_terms = [x**i/factorial(i)]

    while taylor_terms[-1] >= eps:
        i += 1
        taylor_terms.append(x**i/factorial(i))

    return sum(taylor_terms), i
```

Advanced topic: Recursion

- In order to understand recursion, one must first understand recursion

Advanced topic: Recursion

- In order to understand recursion, one must first understand recursion
- A recursive function includes a call to itself (a function within a function)

Advanced topic: Recursion

- In order to understand recursion, one must first understand recursion
- A recursive function includes a call to itself (a function within a function)
  - This could lead to infinite calls;
  - A base case is required so that recursion is stopped;
  - Base case does not call itself, simply returns.

Advanced topic: Recursion

- In order to understand recursion, one must first understand recursion

Advanced topic: Recursion

- In order to understand recursion, one must first understand recursion
- A recursive function includes a call to itself (a function within a function)

Advanced topic: Recursion

- In order to understand recursion, one must first understand recursion
- A recursive function includes a call to itself (a function within a function)
  - This could lead to infinite calls;
  - A base case is required so that recursion is stopped;
  - Base case does not call itself, simply returns.

Recursion: example

```
def mystery(a, b):
  if b == 1:
     # Base case
     return a
  else:
     # Recursive function call
     return a + mystery(a, b-1)
```

Recursion: example

```
def mystery(a, b):
  if b == 1:
     # Base case
     return a
  else:
     # Recursive function call
     return a + mystery(a, b-1)
```

- What does this function do?

Recursion: example

```
def mystery(a, b):
  if b == 1:
     # Base case
     return a
  else:
     # Recursive function call
     return a + mystery(a, b-1)
```

- What does this function do?
- Can you spot the error?

Recursion: example

```
def mystery(a, b):
  if b == 1:
     # Base case
     return a
  else:
     # Recursive function call
     return a + mystery(a, b-1)
```

- What does this function do?
- Can you spot the error?
- How deep can you go? Which values of b don't work anymore?

Practice

Define a function that computes the factorial of a number, $n!$, using recursion:

$$n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$$

Practice

Define a function that computes the factorial of a number, $n!$, using recursion:

$$n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$$

```
def factorial(n):
    """Compute the factorial of n"""
    assert isinstance(n,int) and n >= 0, 'Use positive integers only'

    if n==1:
      return 1
    else:
      return n*factorial(n-1)
```

Scope of functions and variables in Python

In Python, the scope of a variable refers to the regions of a program where that variable is accessible. Understanding the scope of variables helps to avoid bugs and maintain a clean codebase. The scopes in Python are categorized as follows:

Local Scope Variables defined inside a function are in the local scope of that function. They can only be accessed within that function.

Enclosing Scope In the case of nested functions, a function will have access to the variables of the functions it is nested within.

Global Scope Variables defined at the top-level of a script are global and can be accessed by all functions in the script, unless overridden within a function.

Built-in Scope Python has a number of built-in identifiers that should not be used as variable names as they have special significance. Examples include *print*, *list*, *dict*, etc.

Examples Variable Scope

1. Local Scope:

```
def my_func():
local_var = 100 # Local scope
   print(local_var)
```

2. Enclosing Scope:

```
def outer_func():
   outer_var = 200 # Enclosing scope

   def inner_func():
      print(outer_var)

   inner_func()
```

3. Global Scope:

```
global_var = 300 # Global scope

def another_func():
   print(global_var)
```

4. Built-in Scope:

```
print(max, min, len, str, int, list)
```

Exercise Variable Scope

Investigate the behavior of the following nested functions and variables with the same name:

```
def outer_func():
    outer_var = 200

    def inner_func():
        outer_var = 500
        print(outer_var)

    inner_func()
```

Lambda functions (1)

Consider the mathematical function $f(x) = x^2 + e^x$ defined as a Python function block:

```
1 def f(x):
2     from math import exp
3     return x**2 + np.exp(x)
```

Note:

- The function is defined using the `def` keyword.
- The variables and `exp` function used are defined *locally*. They will not be available globally unless defined as such.
- The function is defined in a Python script, not in a separate file.

Lambda functions (2)

If you do not want to create a new function block, you can create an *lambda function*: Lambda functions are small, anonymous functions that can be instantiated in a single line, or even as an argument to a function.

```
1 from math import exp
2 f = lambda x: x**2 + exp(x)
```

Lambda functions (2)

If you do not want to create a new function block, you can create an *lambda function*: Lambda functions are small, anonymous functions that can be instantiated in a single line, or even as an argument to a function.

```
from math import exp
f = lambda x: x**2 + exp(x)
```

- `f`: the name of the function
- `lambda`: used to define the inline function
- `x`: the input argument (can be multiple, comma separated)
- `:`: colon indicating the function definition will start
- `x**2 + exp(x)`: the actual function

Lambda functions (2)

If you do not want to create a new function block, you can create an *lambda function*: Lambda functions are small, anonymous functions that can be instantiated in a single line, or even as an argument to a function.

```
from math import exp
f = lambda x: x**2 + exp(x)
```

- `f`: the name of the function
- `lambda`: used to define the inline function
- `x`: the input argument (can be multiple, comma separated)
- `:`: colon indicating the function definition will start
- `x**2 + exp(x)`: the actual function

```
xsqr_exp = [f(x) for x in range(5)]
print(xsqr_exp)
```

```
[1.0, 3.718281828459045, 11.38905609893065, 29.085536923187668, 70.59815003314424]
```

Today's outline

Using Modules in Python (1)

Modules are files containing Python code, used to organize functionalities and reuse code across projects. To use a module, it must first be imported using the `import` keyword. Here, we import the entire `math` module:

```
>>> import math
```

Using Modules in Python (1)

Modules are files containing Python code, used to organize functionalities and reuse code across projects. To use a module, it must first be imported using the `import` keyword. Here, we import the entire `math` module:

```
>>> import math
```

Once imported, use the dot notation to access functions and variables defined in the module:

```
>>> math.sqrt(16)
4.0
```

Using Modules in Python (1)

Modules are files containing Python code, used to organize functionalities and reuse code across projects. To use a module, it must first be imported using the `import` keyword. Here, we import the entire `math` module:

```
>>> import math
```

Once imported, use the dot notation to access functions and variables defined in the module:

```
>>> math.sqrt(16)
4.0
```

You can import specific attributes from a module using the `from ... import ...` syntax:

```
>>> from math import sqrt
>>> sqrt(16)
4.0
```

Using Modules in Python (2)

Alias module names using the `as` keyword to shorten module names and avoid naming conflicts:

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

Using Modules in Python (2)

Alias module names using the `as` keyword to shorten module names and avoid naming conflicts:

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

To view the list of all functions and variables in a module, use the `dir()` function:

```
>>> import math
>>> dir(math)
```

Using Modules in Python (2)

Alias module names using the `as` keyword to shorten module names and avoid naming conflicts:

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

To view the list of all functions and variables in a module, use the `dir()` function:

```
>>> import math
>>> dir(math)
```

Get help on how to use a module or a function using the `help()` function:

```
>>> help(math.sqrt)
```

The math module

Many mathematical operations and concepts are available in the math module:

```
from math import pi,sin,sqrt,log10\
,exp,floor,ceil,factorial,inf,log
print(pi)
print(sin(0.2*pi))
print(sqrt(2))
print(log10(10_000))
print(exp(1))
print(log(exp(2)))
print(floor(2.57))
print(ceil(2.57))
print(floor(-2.57))
print(round(2.4))
print(round(4.5))
print(factorial(5))
print(f"1 divided by infinity equals {1/inf}")
```

The math module

Many mathematical operations and concepts are available in the math module:

```
from math import pi,sin,sqrt,log10\
,exp,floor,ceil,factorial,inf,log
print(pi)
print(sin(0.2*pi))
print(sqrt(2))
print(log10(10_000))
print(exp(1))
print(log(exp(2)))
print(floor(2.57))
print(ceil(2.57))
print(floor(-2.57))
print(round(2.4))
print(round(4.5))
print(factorial(5))
print(f"1 divided by infinity equals {1/inf}")
```

```
3.141592653589793
0.5877852522924731
1.4142135623730951
4.0
2.718281828459045
2.0
2
3
-3
2
4
120
1 divided by infinity equals 0.0
```

Practice

Use the math module to compute $y = \sin(x)$ for 9 equidistant points $x \in [0, 2\pi]$, including end-points.

- Use *list comprehensions* to generate the lists `x` and `y`

Practice

Use the math module to compute $y = \sin(x)$ for 9 equidistant points $x \in [0, 2\pi]$, including end-points.

- Use *list comprehensions* to generate the lists `x` and `y`

```
1 from math import pi,sin
2 x_values = [x/8*2*pi for x in range(9)]
3 y_values = [sin(x) for x in x_values]
4
5 for x,y in zip(x_values,y_values):
6    print(f"{x: 10.4f},{y: 10.4f}")
```

Practice

Use the math module to compute $y = \sin(x)$ for 9 equidistant points $x \in [0, 2\pi]$, including end-points.

- Use *list comprehensions* to generate the lists `x` and `y`

```
1 from math import pi,sin
2 x_values = [x/8*2*pi for x in range(9)]
3 y_values = [sin(x) for x in x_values]
4
5 for x,y in zip(x_values,y_values):
6     print(f"{x: 10.4f},{y: 10.4f}")
```

```
0.0000, 0.0000
0.7854, 0.7071
1.5708, 1.0000
2.3562, 0.7071
3.1416, 0.0000
3.9270, -0.7071
4.7124, -1.0000
5.4978, -0.7071
6.2832, -0.0000
```

The random module (1)

Random number generators and sampling tools are available through the random module. A few examples for integers and sequences:

```
import random as rnd

# Random integers
random_integers = [rnd.randint(0,10) for i in range(10)]
print(f'{random_integers = }')

# Sample from given population
my_range = range(12) # [0, 1, 2, ..., 11]
select_from_range = rnd.sample(my_range,8)
print(f'{select_from_range = }') # Selected 8 elements

# Choose 1 element from list
days_of_week = ['Monday','Tuesday','Wednesday','Thursday','Friday','Saturday','Sunday']
day = rnd.choice(days_of_week)
print(f"I've chosen {day} as my lucky day!")
```

The random module (1)

Random number generators and sampling tools are available through the random module. A few examples for integers and sequences:

```
import random as rnd

# Random integers
random_integers = [rnd.randint(0,10) for i in range(10)]
print(f'{random_integers = }')

# Sample from given population
my_range = range(12) # [0, 1, 2, ..., 11]
select_from_range = rnd.sample(my_range,8)
print(f'{select_from_range = }') # Selected 8 elements

# Choose 1 element from list
days_of_week = ['Monday','Tuesday','Wednesday','Thursday','Friday','Saturday','Sunday']
day = rnd.choice(days_of_week)
print(f"I've chosen {day} as my lucky day!")
```

```
random_integers = [6, 6, 2, 5, 5, 8, 3, 7, 3, 4]
select_from_range = [2, 3, 10, 1, 6, 4, 8, 5]
I've chosen Wednesday as my lucky day!
```

The random module (2)

Examples for real valued distributions:

```
import random as rnd

# Random number (uniform distribution) 0 < x < 1
x = [rnd.random() for i in range(5)]
print(x)

# Random number (uniform distribution) between given bounds
x = [rnd.uniform(1,3) for i in range(5)]
print(x)

# Random number from a Gauss distribution (mu=0, sigma=2)
x = [rnd.gauss(0,2) for i in range(5)]
print(x)
```

The random module (2)

Examples for real valued distributions:

```
import random as rnd

# Random number (uniform distribution) 0 < x < 1
x = [rnd.random() for i in range(5)]
print(x)

# Random number (uniform distribution) between given bounds
x = [rnd.uniform(1,3) for i in range(5)]
print(x)

# Random number from a Gauss distribution (mu=0, sigma=2)
x = [rnd.gauss(0,2) for i in range(5)]
print(x)
```

```
[0.6697878114597362, 0.4136014290205997, 0.5108247513505662, 0.44260043089156076, 0.9902269207988261]
[2.4749381508841077, 2.943448233960596, 2.516639180020423, 1.0481550073898795, 1.961356325508141]
[1.8199856149229392, 2.000097897306016, -2.4604868187736026, -0.46836605162997846, -2.5069012642608803]
```

Practice

- Create a *function* that returns a list of *N* dice throws (cubic dice, values 1-6)
- Throw the dice many times
- Print for each value how often it has been thrown

Practice

- Create a *function* that returns a list of *N* dice throws (cubic dice, values 1-6)
- Throw the dice many times
- Print for each value how often it has been thrown

```
1  import random as rnd
2
3  def throw_dice(N):
4     return [rnd.randint(1,6) for _ in range(N)]
5
6  throws = throw_dice(40)
7  print(throws)
8
9  for i in range(1,7):
10    n = len([t for t in throws if t==i])
11    print(f"Value {i} was thrown {n} times")
```

Practice

- Create a *function* that returns a list of *N* dice throws (cubic dice, values 1-6)
- Throw the dice many times
- Print for each value how often it has been thrown

```python
import random as rnd

def throw_dice(N):
    return [rnd.randint(1,6) for _ in range(N)]

throws = throw_dice(40)
print(throws)

for i in range(1,7):
    n = len([t for t in throws if t==i])
    print(f"Value {i} was thrown {n} times")
```

```
[5, 1, 1, 4, 6, 3, 3, 1, 5, 6, 1, 1, 1, 1, 5, 5, 2, 1, 1, 2, 2, 5, 5, 4, 6, 1, 3, 5, 6, 3, 1, 5, 6, 2, 3, 1, 6, 3, 2, 1]
Value 1 was thrown 13 times
Value 2 was thrown 5 times
Value 3 was thrown 6 times
Value 4 was thrown 2 times
Value 5 was thrown 8 times
Value 6 was thrown 6 times
```

Today's outline

In conclusion...

- Python: A versatile development language. Easy to use libraries makes this language multi-purpose and easy to use.
- Programming basics: variables, operators and functions, locality of variables, modules and recursive operations

In conclusion...

- Python: A versatile development language. Easy to use libraries makes this language multi-purpose and easy to use.
- Programming basics: variables, operators and functions, locality of variables, modules and recursive operations
- For now: exercises on slide deck and Python modules

In conclusion...

- Python: A versatile development language. Easy to use libraries makes this language multi-purpose and easy to use.
- Programming basics: variables, operators and functions, locality of variables, modules and recursive operations
- For now: exercises on slide deck and Python modules

Practice vectors and arrays

1. Create a list `x` with the elements:
   - `[2, 4, 6, 8, ..., 16]`
   - `[0, 0.5, 2/3, 3/4, ..., 99/100]`
2. Create a list `x` with the elements: $x_n = \frac{(-1)^n}{2n-1}$ for $n = 1, 2, 3, \ldots, 200$. Find the sum of the first 50 elements $x_1, \ldots, x_{50}$.
3. Let `x = list(range(20, 201, 10))`. Create a list `y` of the same length as `x` such that:
   - `y[i] = x[i] - 3`
   - `y[i] = x[i]` for every even index *i* and `y[i] = x[i] + 11` for every odd index *i*.
4. Let `T = np.array([[3, 4, 6], [1, 8, 6], [-4, 3, 6], [5, 6, 6]]).T`. Perform the following operations on T:
   - Retrieve a list consisting of the 2nd and 4th elements of the 3rd row.
   - Find the minimum of the 3rd column.
   - Find the maximum of the 2nd row.
   - Compute the sum of the 2nd column
   - Compute the mean of the row 1 and the mean of row 3

Practice plotting

1. Plot the functions $f(x) = x$, $g(x) = x^3$, $h(x) = e^x$ and $z(x) = e^{x^2}$ over the interval $[0, 4]$ on the normal scale and on the log-log scale. Use an appropriate sampling to get smooth curves. Describe your plots by using the functions: `plt.xlabel`, `plt.ylabel`, `plt.title` and `plt.legend`.

2. Make a plot of the functions: $f(x) = sin(1/x)$ and $g(x) = cos(1/x)$ over the interval $[0.01, 0.1]$. How do you create `x` so that the plots look sufficiently smooth?

Practice control flow and loops (1)

1. Write a function that uses two logical input arguments with the following behaviour:

$$f(\text{true}, \text{true}) \mapsto \text{false}$$
$$f(\text{false}, \text{true}) \mapsto \text{true}$$
$$f(\text{true}, \text{false}) \mapsto \text{true}$$
$$f(\text{false}, \text{false}) \mapsto \text{false}$$

2. Write a function that computes the factorial of x:

$$f(x) = x! = 1 \times 2 \times 3 \times 4 \times \ldots \times x$$

- Using a loop-construction
- Using recursion

Practice control flow and loops (2)

1. Write a function that computes the exponential function using the Taylor series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

   until the last term is smaller than 10^{-6}.

2. Use a script to compute the result of the following series:

$$f_n = \sum_{n=1}^{\infty} \frac{1}{\pi^2 n^2}$$

   This should give you an indication of the fraction this series converges to.

   - Now plot in two vertically aligned subplots i) The result as a function of n, and ii) the difference with the earlier mentioned fraction as a function of n. For the latter, consider carefully the axis scale!

Practice logical indexing

1. Let `x = np.linspace(-4,4,1000)`, $y_1 = 3x^2 - 4x - 6$ and $y_2 = 1.5x - 1$. Use logical indexing to determine function $y_3 = \max(\max(y_1, y_2), 0)$. Plot the function.
2. Consider these data concerning the age (in years), length (in cm) and weight (in kg) of twelve adult men: `A = [41 25 33 29 64 34 47 38 49 32 26 26]; H = [165 186 177 190 156 174 164 205 184 190 165 171]; W = [75 90 97 60 74 65 101 85 91 75 87 70];`.
   - Calculate the average of all vectors (age, weight and length).
   - Combine the command `length` with logical indexing to determine how many men in the group are taller than 182 cm.
   - What is the average age of men with a body-mass index ($B \equiv \frac{W}{L^2}$ with W in kg and L in m) larger than 25? And for men with a $B < 25$?
   - How many men are older than the average and at the same time have a BMI below 25?

Practice algorithm: Fourier series for heat equation

The unsteady 1D heat equation in 1D in a slab of material is given as:

$$\frac{\partial T}{\partial t} = k\frac{\partial^2 T}{\partial x^2}$$

We can express the temperature profile $T(x,t)$ in the slab using a Fourier sine series. For an initial profile T(x,0) = 20 and fixed boundary values T(0,t) = T(L,t) = 0, the solution is given as:

$$T(x,t) = \sum_{n=1}^{n=\infty} \frac{40(1-(-1)^n)}{n\pi} \sin\left(\frac{n\pi x}{L}\right) \exp\left(-kt\frac{n\pi}{L}^2\right)$$

- Create a script to solve this equation using loops and/or conditional statements

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

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 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
```

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
```

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
```

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
```

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)
```

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)
```

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)
```

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

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`:

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

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

```
import time

def f(x):
   return x**2+2*x-4

long_data = np.linspace(0,20,10_000_000)

start = time.time()
y1 = f(long_data)
print(f"The vectorized approach took {time.time()-start} seconds")

y2 = np.zeros_like(long_data)
start = time.time()
for i in range(len(long_data)):
   y2[i] = f(long_data[i])
print(f"The iterated approach took {time.time()-start} seconds")
```

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]
```

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

- 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 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`.

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
```

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()
```

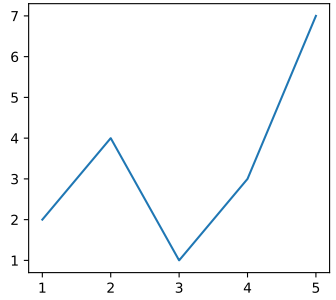
Introduction to Matplotlib

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

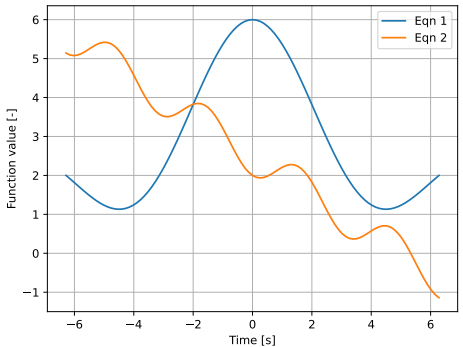
Adding multiple plots, axis labels and a legend:

```
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()
```

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

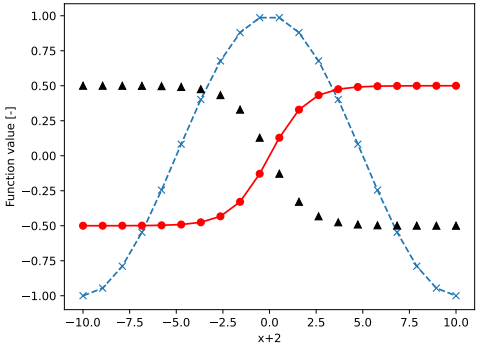
Changing markers, line styles:

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  t = np.linspace(-10.0, 10.0, 20)
5  p = np.cos(np.pi * t/10)
6  q = 1/(1+np.exp(-t)) - 0.5
7
8  plt.plot(t,p,'--x') # Dashed line, cross
9  plt.plot(t,q,'r-o') # Red, line, circles
10 plt.plot(t,-q,'k^') # Black triangle marker
11 plt.xlabel('x+2')
12 plt.ylabel('Function value [-]')
13 plt.show()
```

Line plot styles

Changing markers, line styles:

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  t = np.linspace(-10.0, 10.0, 20)
5  p = np.cos(np.pi * t/10)
6  q = 1/(1+np.exp(-t)) - 0.5
7
8  plt.plot(t,p,'--x') # Dashed line, cross
9  plt.plot(t,q,'r-o') # Red, line, circles
10 plt.plot(t,-q,'k^') # Black triangle marker
11 plt.xlabel('x+2')
12 plt.ylabel('Function value [-]')
13 plt.show()
```



Subplots

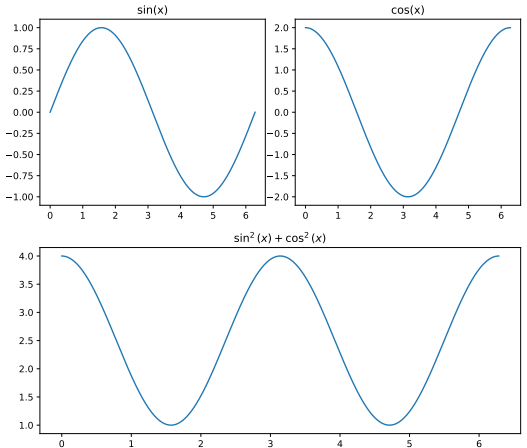
Multiple plots in a single figure are made by specifying a grid using subplot:

```
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()
```

Subplots

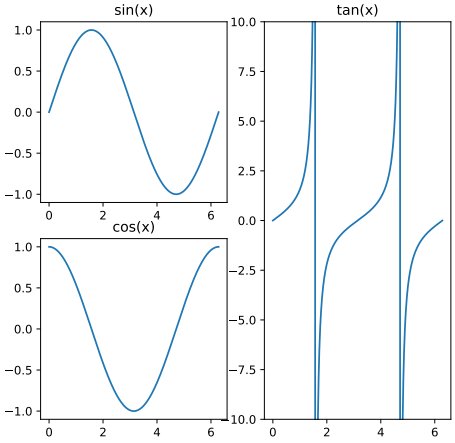
Multiple plots in a single figure are made by specifying a grid using subplot:

```
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()
```



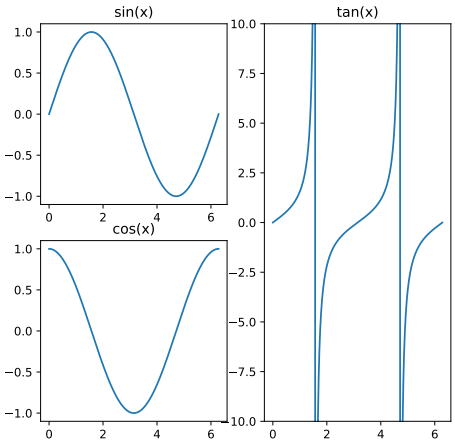
Practice

Try to create the following figure:



Practice

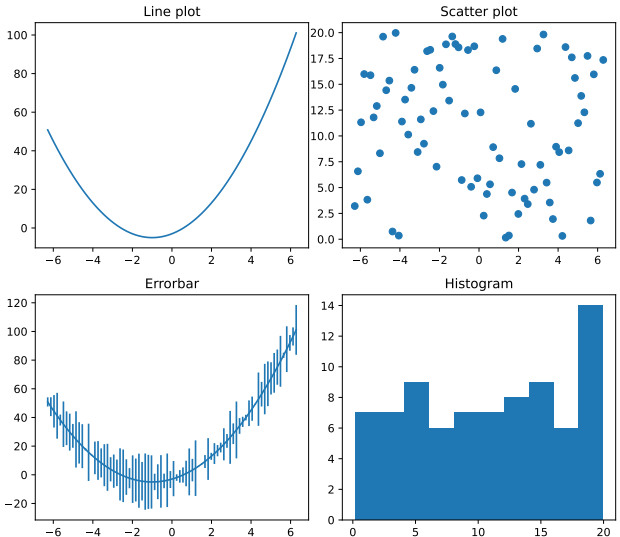
Try to create the following figure:




```python
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0,2*np.pi,1000)
y1,y2,y3 = np.sin(x), np.cos(x), np.tan(x)

ax1 = plt.subplot(2, 2, 1)
ax1.plot(x,y1)
ax1.set_title('sin(x)')

ax2 = plt.subplot(2, 2, 3)
ax2.plot(x,y2)
ax2.set_title('cos(x)')

# Second index of 2x1 grid (whole bottom)
ax3 = plt.subplot(1, 2, 2)
ax3.plot(x,y3)
ax3.set_title('tan(x)')
ax3.set_ylim(-10, 10)
```

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')

```

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)

```

Surface plot

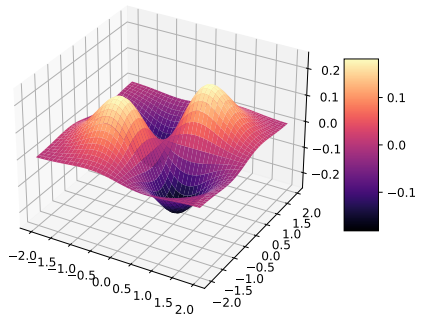
We draw a surface plot of $f(x,y) = xye^{(-x^2-y^2)}$:

```
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)

```

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

- 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

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 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 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")
```

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.

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)
```

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")
```

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

- 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

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!

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!

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!

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!

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!

Make it work

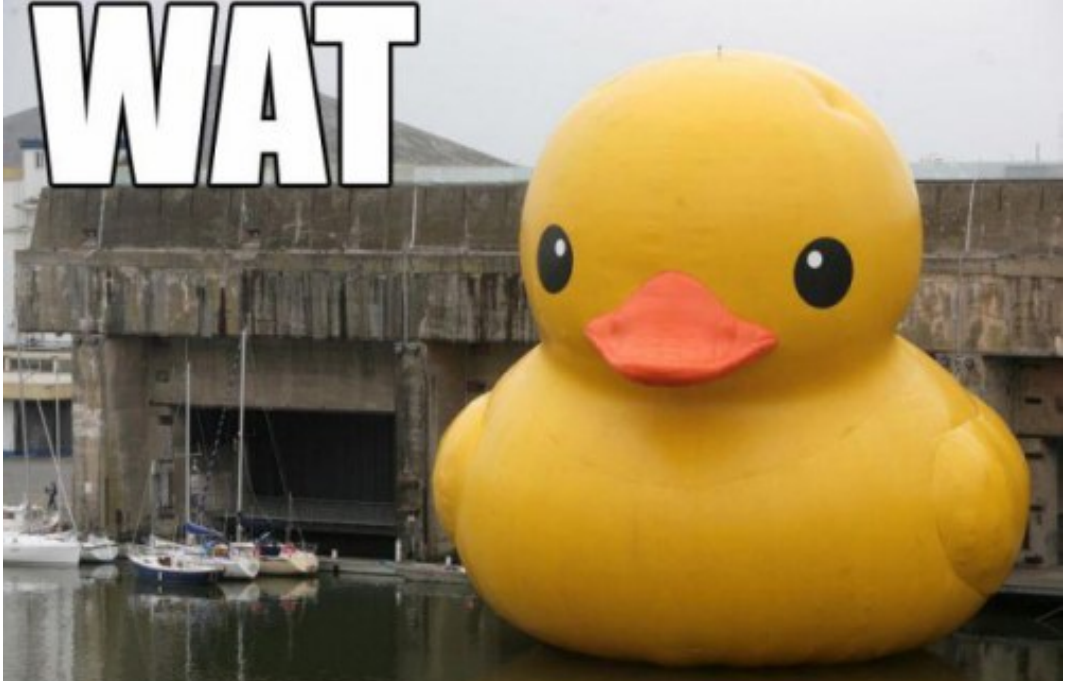
Use the building blocks of previous lecture to create an algorithm:

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:

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.

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:

```
1 # Start program
```

How not to comment

- Useless:

```
# Start program
```

- Obvious:

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

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...
```

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.

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.

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

- 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

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

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.

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.

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

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.

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.

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.

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.

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 $F_2 = 1$

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 $F_2 = 1$

```
def fibonacci_recursive(N):
    """
    Prints out the Nth Fibonacci number to the screen.
    SYNTAX: fibonacci_recursive(N)
    """
    if N > 2:
        Nminus1 = fibonacci_recursive(N-1)
        Nminus2 = fibonacci_recursive(N-2)
        out = Nminus1 + Nminus2
    elif N == 1 or N == 2:
        out = 1
    else:
        raise ValueError('Input argument was invalid')
    return out
```

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.

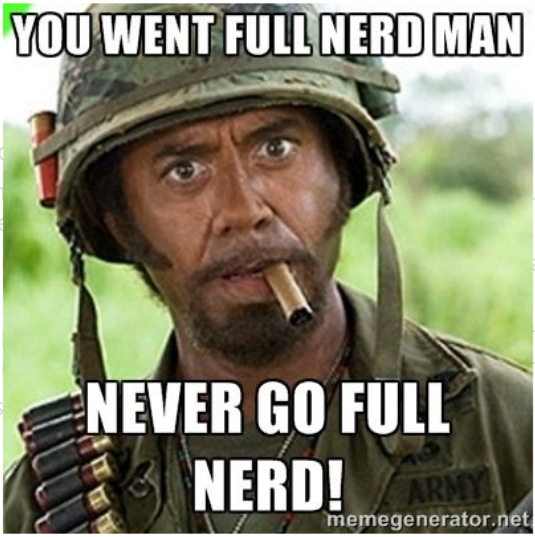
Today's outline

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.

Advanced concepts

- Object oriented pr
- Memory managem u to allocate computer memory yourself (e
- External libraries: i al functionality you are looking for
- Compiling and scri ng a program to computer-languag is on the fly.
- Parallellization: Dis rocessors 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.

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.

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.