

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 (6E5X0), 2023-2024

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

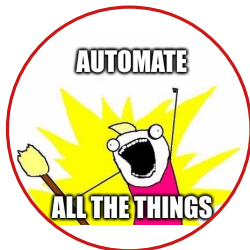
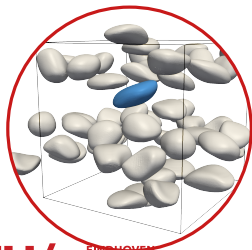
- Using modules
- Math module
- The random module

● Conclusions

● Exercises

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?

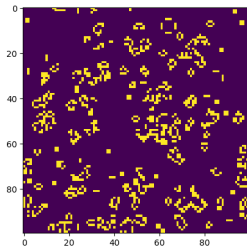
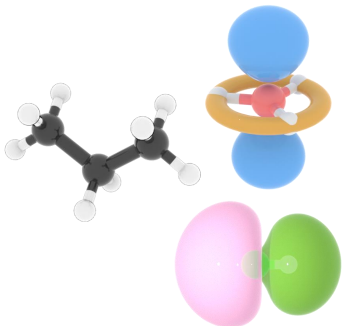
A program is a sequence of instructions that is written to perform a certain task on a computer.

- 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)

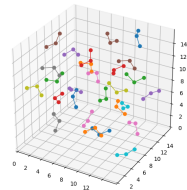
Versatility of Python



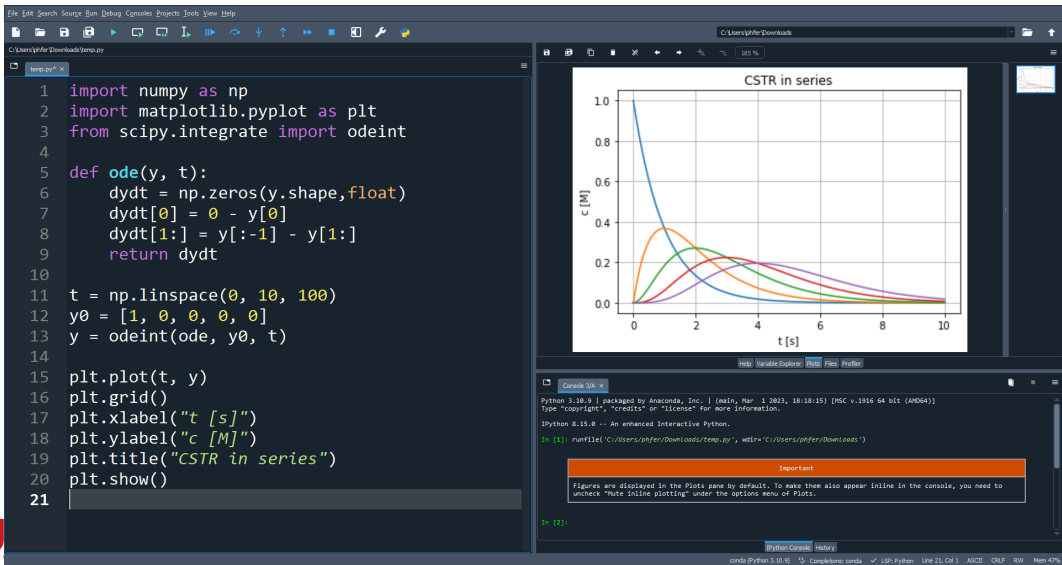
```

1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 import numpy as np
5 import sys
6
7 from PIL import Image, ImageFont, ImageDraw
8 from matplotlib import pyplot as plt
9 import time
10
11
12 # Parameters
13 # =====
14 # Image size
15 width = 100
16 height = 100
17 # Image format
18 format = 'png'
19 # Output file
20 output_file = 'image.png'
21 # Image name
22 image_name = 'image.png'
23
24 # Create image
25 def create_image():
26     # Create a blank image
27     img = Image.new('RGB', (width, height), color='black')
28     # Draw a grid of pixels
29     for x in range(width):
30         for y in range(height):
31             # Draw a pixel
32             img.putpixel((x, y), (255, 255, 255))
33     # Save the image
34     img.save(output_file)
35
36 # Main
37 if __name__ == '__main__':
38     create_image()
39 
```

$$\frac{\partial^2 c_{ijk}}{\partial z^2} \approx \text{Div}_{ii'} \text{Grad} z_{i'j''} c_{j''jk} + \text{Div}_{ii'}$$

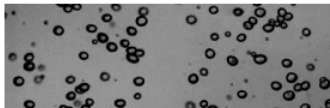
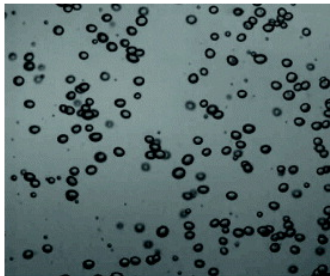


Versatility of Python: ODE solver



Versatility of Python: Image analysis

```
1 # Importing necessary libraries
2 import numpy as np
3 from scipy import ndimage
4 from PIL.Image import fromarray
5 from skimage import io, color, feature, measure
6
7 # Loading and processing image
8 I = io.imread('bub0.png')
9 BW = color.rgb2gray(I)
10 E = feature.canny(BW)
11 F = ndimage.binary_fill_holes(E)
12
13 # Show final image
14 fromarray(F).show()
```



Versatility of Python: Curve fitting

The screenshot displays a Python IDE with a script for curve fitting and its corresponding plot.

Script (temp1.py):

```

1 import numpy as np
2 from scipy.optimize import curve_fit
3 import matplotlib.pyplot as plt
4
5 # Define the form of the function you want to fit
6 def func(x, a, b, c):
7     return a * x**2 + b * x + c
8
9 # Generate example data
10 x = np.linspace(-10, 10, 100)
11 y = func(x, 1, -2, 3) + np.random.normal(scale=10, size=100)
12
13 # Use curve_fit to fit the function to the data
14 popt, _ = curve_fit(func, x, y)
15
16 # Generate y-data based on the fit
17 y_fit = func(x, *popt)
18
19 # Create a pretty plot
20 plt.figure(figsize=(8, 6))
21 plt.plot(x, y, 'o', mfc='none', label='Data')
22 plt.plot(x, y_fit, '--', label='Fit')
23 plt.xlabel('X')
24 plt.ylabel('Y')
25 plt.title('Curve Fitting y = a * x^2 + b * x + c')
26 plt.grid(True)
27 plt.legend()
28 plt.show()
29

```

Plot: The plot shows a scatter of data points (blue circles) and a fitted curve (orange dashed line). The title is "Curve Fitting y = a * x^2 + b * x + c". The x-axis is labeled "X" and ranges from -10.0 to 10.0. The y-axis is labeled "Y" and ranges from -20 to 120. A legend in the top right corner identifies the blue circles as "Data" and the orange dashed line as "Fit".

Console: The console shows the execution of the script, including the version of Python (3.10.9) and the location of the script file.

Important Note: A message box states: "Figures are displayed in the Plots pane by default. To make them also appear inline in the console, you need to uncheck 'Hide inline plotting' under the options menu of Plots."

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
- Typing part of a command and pressing  completes the command and lists the possibilities
- If a computation takes too long, you can press  +  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, Siau and Bayen
- Search the web, Reddit, YouTube, etc.

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

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)

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)`:

```
1 more_brands.append('Volvo') # Append object (here: string literal) at the end of the list
2 more_brands.insert(1, 'BMW') # Insert object at index 1
3 more_brands.sort()          # Sorts the list in-place
4 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:

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

```
3, 6
8885416 8885544
```

```
1 >>> list_a = ['aa', 1, 'bb', 12, True, 1.618]
2 >>> list_b = list_a
3 >>> list_b[2] = 'cc'
4 >>> print(list_a, list_b, sep='\n')
5 >>> 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
- 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

```
1 >>> 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.

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

```
1 >>> 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.

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.

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

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}')
```

```
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
```

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.
- 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.

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
```

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.

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]
```

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
```

- 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.

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

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}!")
```

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)
```

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
- **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 i = 1
2 l = ['a', 'b']
3
4 func(i, l)
5
6 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, $n!$:

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

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

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

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
- 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

```
1 def mystery(a, b):  
2     if b == 1:  
3         # Base case  
4         return a  
5     else:  
6         # Recursive function call  
7         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:

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

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:

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

2. Enclosing Scope:

```
1 def outer_func():  
2     outer_var = 200 # Enclosing scope  
3  
4     def inner_func():  
5         print(outer_var)  
6  
7     inner_func()
```

3. Global Scope:

```
1 global_var = 300 # Global scope  
2  
3 def another_func():  
4     print(global_var)
```

4. Built-in Scope:

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

Exercise Variable Scope

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

```
1 def outer_func():
2     outer_var = 200
3
4     def inner_func():
5         outer_var = 500
6         print(outer_var)
7
8     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)
```

- `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

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

```
1 [1.0, 3.718281828459045, 11.38905609893065, 29.085536923187668, 70.59815003314424]
```

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

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
```

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**:

```

1 from math import pi, sin, sqrt, log10\
2 , exp, floor, ceil, factorial, inf, log
3 print(pi)
4 print(sin(0.2*pi))
5 print(sqrt(2))
6 print(log10(10_000))
7 print(exp(1))
8 print(log(exp(2)))
9 print(floor(2.57))
10 print(ceil(2.57))
11 print(floor(-2.57))
12 print(round(2.4))
13 print(round(4.5))
14 print(factorial(5))
15 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 8 equidistant points $x \in [0, 2\pi]$

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

The random module (1)

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

```

1 import random as rnd
2
3 # Random integers
4 random_integers = [rnd.randint(0,10) for i in range(10)]
5 print(f'{random_integers = }')
6
7 # Sample from given population
8 my_range = range(12) # [0, 1, 2, ..., 11]
9 select_from_range = rnd.sample(my_range,8)
10 print(f'{select_from_range = }') # Selected 8 elements
11
12 # Choose 1 element from list
13 days_of_week = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
14 day = rnd.choice(days_of_week)
15 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**:

```
1 import random as rnd
2
3 # Random number (uniform distribution)  $0 < x < 1$ 
4 x = [rnd.random() for i in range(5)]
5 print(x)
6
7 # Random number (uniform distribution) between given bounds
8 x = [rnd.uniform(1,3) for i in range(5)]
9 print(x)
10
11 # Random number from a Gauss distribution (mu=0, sigma=2)
12 x = [rnd.gauss(0,2) for i in range(5)]
13 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

```
[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

● 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

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:
 - $[2, 4, 6, 8, \dots, 16]$
 - $[0, 0.5, 2/3, 3/4, \dots, 99/100]$
- ② Create a list x with the elements: $x_n = \frac{(-1)^n}{2n-1}$ for $n = 1, 2, 3, \dots, 200$. Find the sum of the first 50 elements x_1, \dots, x_{50} .
- ③ Let $x = \text{list}(\text{range}(20, 201, 10))$. 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 $T = \text{np.array}([[3, 4, 6], [1, 8, 6], [-4, 3, 6], [5, 6, 6]])$. 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 4

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 :

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

- 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 $x = \text{np.linspace}(-4, 4, 1000)$, $y_1 = 3x^2 - 4x - 6$ 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:

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

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