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

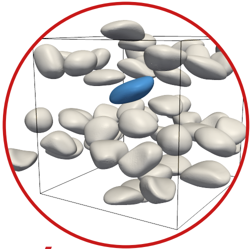
EINDHOVEN UNIVERSITY OF TECHNOLOGY

Today's outline

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Why should you learn something about programming?

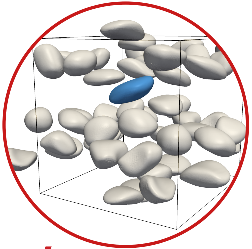
- Scientific analyses depend more than ever on computer programs and simulation methods

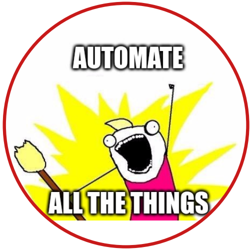


TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Why should you learn something about programming?

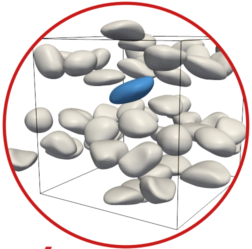
- Scientific analyses depend more than ever on computer programs and simulation methods
- Knowledge of programming allows you to automate routine tasks

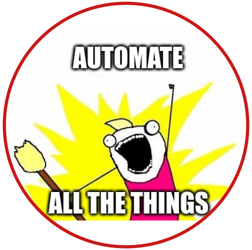




Why should you learn something about programming?

- 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

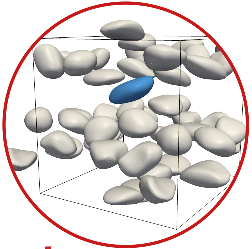


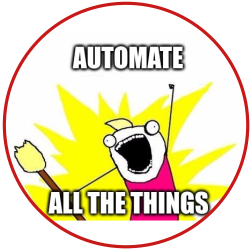


EINDHOVEN UNIVERSITY OF TECHNOLOGY

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
- Learn to think by dissecting a problem into smaller, easier to solve, parts





Introduction to programming

What is a program?

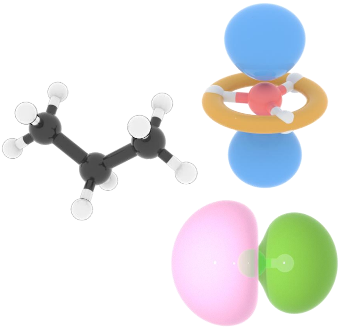
A program is a sequence of instructions that is written to perform a certain task on a computer.

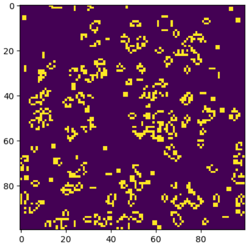
- The computation might be something mathematical, a symbolic operation, image analysis, etc.

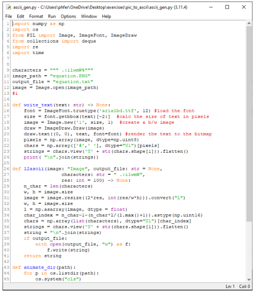
Program layout

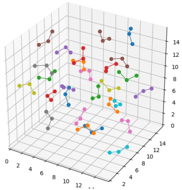
1. Input (Get the radius of a circle)
2. Operations (Compute and store the area of the circle)
3. Output (Print the area to the screen)

Versatility of Python

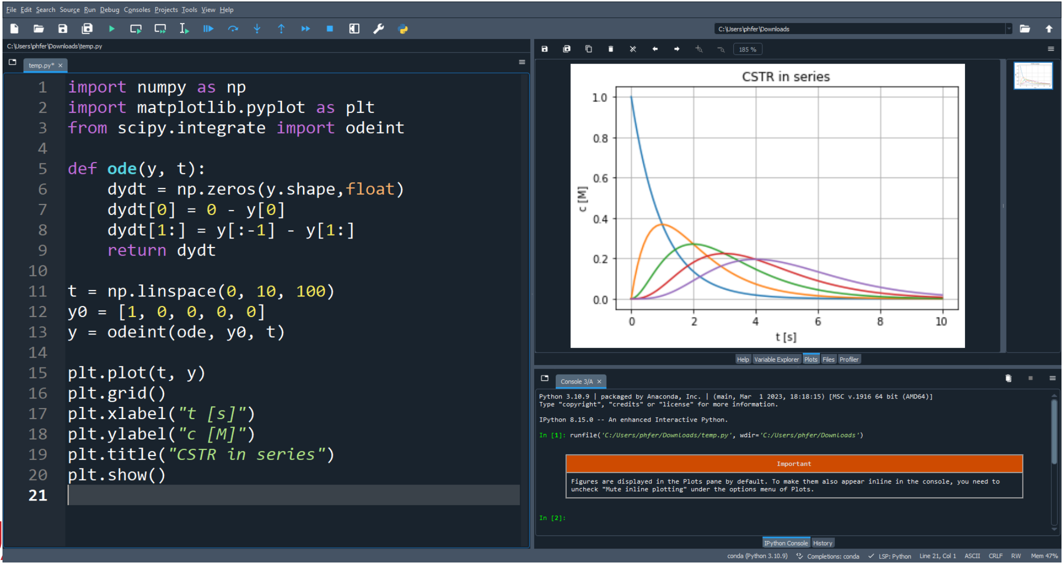






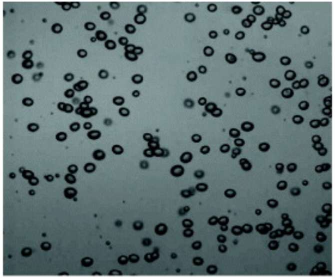


Versatility of Python: ODE solver



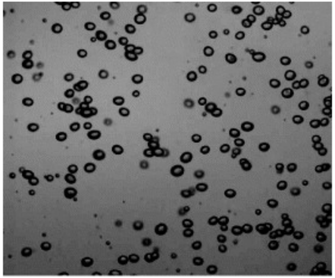
Versatility of Python: Image analysis

```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()
```



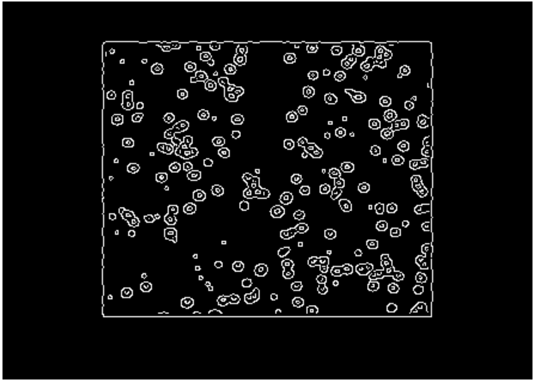
Versatility of Python: Image analysis

```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()
```



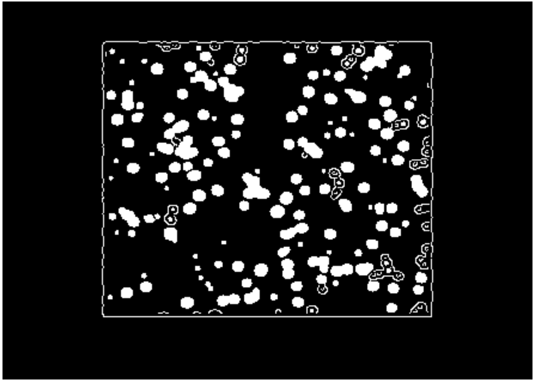
Versatility of Python: Image analysis

```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()
```



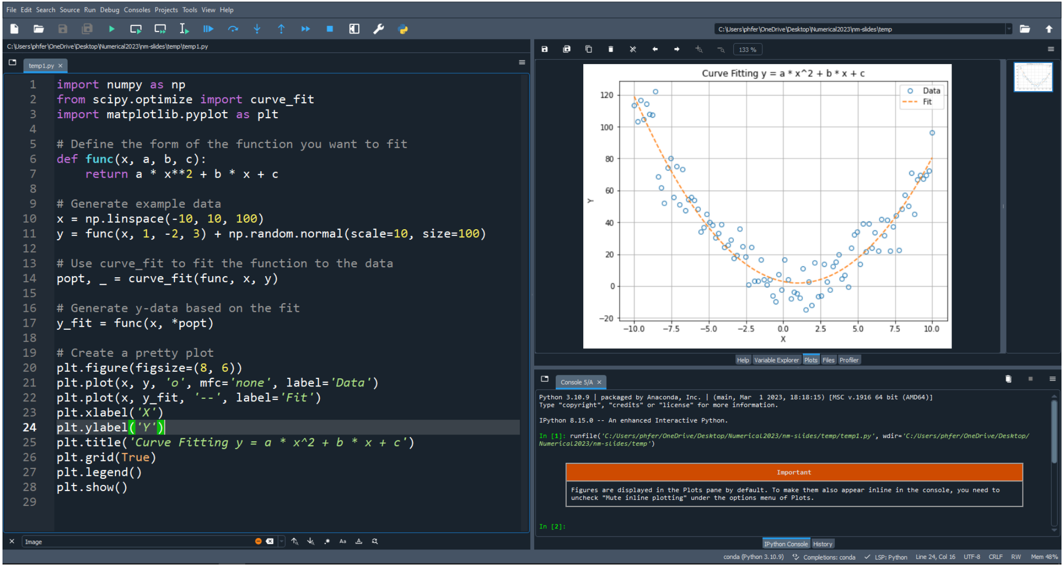
Versatility of Python: Image analysis

```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()
```



EINDHOVEN UNIVERSITY OF TECHNOLOGY

Versatility of Python: Curve fitting



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!")`).

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

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
```

EINDHOVEN UNIVERSITY OF TECHNOLOGY

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
```

EINDHOVEN UNIVERSITY OF TECHNOLOGY

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
```

EINDHOVEN UNIVERSITY OF TECHNOLOGY

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))
```

EINDHOVEN UNIVERSITY OF TECHNOLOGY

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)]
```

EINDHOVEN UNIVERSITY OF TECHNOLOGY

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.

```
>>> print(d['Charlie'])
>>> d['Alice'] = 25
>>> d['Eve'] = 29
>>> print(d.keys())
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])
```

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

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

EINDHOVEN UNIVERSITY OF TECHNOLOGY

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 [2]:

- 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
```

[2] 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 [2]:

- 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:

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

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

func(i, l)

print(i, l)
```

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

[2] 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.

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Recursion: example

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

Recursion: example

```python
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

```python
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

```python
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:

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

```python
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:

```python
def outer_func():
    outer_var = 200

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

    inner_func()
```

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Lambda functions (1)

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

```
def f(x):
    from math import exp
    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.

```
from math import exp
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.

```python
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:

```python
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:

```python
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`

```
from math import pi,sin
x_values = [x/8*2*pi for x in range(9)]
y_values = [sin(x) for x in x_values]

for x,y in zip(x_values,y_values):
    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`

```python
from math import pi,sin
x_values = [x/8*2*pi for x in range(9)]
y_values = [sin(x) for x in x_values]

for x,y in zip(x_values,y_values):
    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:

```python
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:

```python
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:

```python
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

```
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")
```

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

```
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

EINDHOVEN UNIVERSITY OF TECHNOLOGY

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 $n = 1, 2, 3, \dots, 200$. Find the sum of the first 50 elements $x_1, \dots, x_{50}$.
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:

$$
\begin{aligned}
f(\text{true}, \text{true}) &\mapsto \text{false} \\
f(\text{false}, \text{true}) &\mapsto \text{true} \\
f(\text{true}, \text{false}) &\mapsto \text{true} \\
f(\text{false}, \text{false}) &\mapsto \text{false}
\end{aligned}
$$

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

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

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

EINDHOVEN UNIVERSITY OF TECHNOLOGY