The Programming Language Python

Output

- the upper, grey-shaded part of a "code cell" contains the input (meaning the code that you write) while the lower, white-shaded part contains the output
- in addition to code cells, this "notebook" also contains text cells (like this one)

- it is possible to generate output from code cells in *interactive mode* where only the last output is printed to the output field
- to output more than one line the *script mode* using `print(<message>)` function must be used → accepts any kind of printable output, e.g. strings, lists, numbers

In [1]:
```
"Moin Kiel!" # this isn't printed
"Hello World!"
```

Out[1]:
```
'Hello World!'
```

In [2]:
```
print("Moin Kiel!")
print("Hello World!")  # the famous "Hello World" program
```

```
Moin Kiel!
Hello World!
```

Variables

Definition

- are symbolic names for addressing an area of the main memory → placeholders used to store values
- have *name*, *value*, and *data type* → variable is *declared* and value is *assigned*

Types

- data type of variable determines
  - what values variable can hold,
  - how variable is stored in main memory of computer
- type of variable does not have to be defined like in other programming languages (e.g. Java: `int mynumber;`) → type is deducted from value of variable
- type ran be read using the `type(<var>)` function

In [3]:
```
# str (string): "Hi"
print(type("Hi"))

# int (integer -> whole number): 5
print(type(5))
```

```
# float (floating-point number -> numbers with decimal point): 3.141
print(type(3.141))

# bool (boolean -> True or False): True
print(type(True))
```

```
<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
```

Illegal Names

Some variable names are not allowed. Such are:

- starting with a number
- using illegal characters like '@'
- keywords like `for`

In [4]:
```
1st_illegal_variable = 1
```

```
  Cell In [4], line 1
    1st_illegal_variable = 1
    ^
SyntaxError: invalid decimal literal
```

In [5]:
```
illeg@l_variable = 1
```

```
  Cell In [5], line 1
    illeg@l_variable = 1
    ^
SyntaxError: cannot assign to expression here. Maybe you meant '==' instead of '='?
```

In [6]:
```
for = 1
```

```
  Cell In [6], line 1
    for = 1
        ^
SyntaxError: invalid syntax
```

Case sensitivity

- Python variable names are case sensitive

In [7]:
```
Bob = "Bob"
bob = "bob"

print(Bob == bob) # is 'bob' the same as 'Bob'?
```

```
False
```

Typecasting

- process of converting type of variable - all build-in types in Python can be found here
- done by enclosing variable inside variable type function
  - `str(<var>)` → conversion to string
  - `int(<var>)` → conversion to integer
  - `float(<var>)` → conversion to floating-point
  - `bool(<var>)` → conversion to boolean

- sometimes necessary as certain operation can only work with certain variable types → e.g.only strings can be concatenated, not strings and integers

In [8]:
```
"This is number" + 3
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [8], line 1
----> 1 "This is number" + 3

TypeError: can only concatenate str (not "int") to str
```

In [9]:
```
"This is number " + str(3)
```

Out[9]:
```
'This is number 3'
```

User input

Variables values can be assigned by user input:

- input into program can be provided using the `input(<message>)` function
- input is string datatype
- input can be assigned to variable: `var = input(<message>)`

In [10]:
```
input_variable = input("What is your name?")

print(input_variable)
print(type(input_variable))  # str-type
```

```
Niclas
<class 'str'>
```

Operators

Mathematical operators

- to execute numerical calculations

In [11]:
```
4 + 2  # Addition
```

```
Out[11]: 6
```

```
In [12]: 4 - 2  # Subtraction
```

```
Out[12]: 2
```

```
In [13]: 4 * 2  # Multiplication
```

```
Out[13]: 8
```

```
In [14]: 9 ** 4  # Exponent
```

```
Out[14]: 6561
```

```
In [15]: 9 / 4  # Division: creates floating-point number
```

```
Out[15]: 2.25
```

```
In [16]: 9 // 4  # Integer Division: Division with an integer result. The remainder is of no interest.
```

```
Out[16]: 2
```

```
In [17]: 9 % 4  # Modulus: Calculation of remainder of an integer division?
```

```
Out[17]: 1
```

Assignment (in-place) operators

- mathematical operators to update existing variable values
- generate another number
- an overview of (additional) in-place operators can be found in the official Python documentation

```
In [18]: n = 10
n += 2  # Addition: replaces n = n + 2

print(n)

n = "Hello"  # with strings this method only works for "+" and "+=" - here, it is used to concatenate strings
n += " World"

print(n)
```

```
12
Hello World
```

```
In [19]: n = 10
n -= 2  # Subtraction

print(n)
```

```
8
```

In [20]:
```
n = 10
n *= 2  # Multiplication

print(n)
```
```
20
```

In [21]:
```
n = 10
n **= 2  # Power

print(n)
```
```
100
```

In [22]:
```
n = 10
n /= 2  # Division

print(n)
```
```
5.0
```

In [23]:
```
n = 10
n //= 2  # Integer Division

print(n)
```
```
5
```

In [24]:
```
n = 10
n %= 2  # Modulus

print(n)
```
```
0
```

Relational operators

- generate a boolean value
- can also be chained

In [25]:
```
print(4 > 2)  # greater: also greater-equal '>=', smaller '<', and smaller-equal '<='
print(4 > 2 > 0)  # this is equal to '4 > 2 and 2 > 0' -> this chaining principle also works for other operators
```
```
True
True
```

In [26]:
```
my_int_1 = 4
my_int_2 = 4
my_int_3 = 2

my_float_1 = 4.0
my_float_2 = 4.0
my_float_3 = 2.0

# comparing ints and floats:
print(my_int_1 == my_int_2)
print(my_float_1 == my_float_2)
print()

print(my_int_1 == my_float_1)
print(my_int_3 == my_float_1)
```

```
True
True

True
False
```

- the function combination `hex(id(<object>))` allows checking for the memory address of objects
- objects with the same memory address are aliases ("the same", not only identical in value)

In [27]:
```
# equality vs being the same:
print(my_int_1 is my_int_2)  # comparison for being same only works with ints, not with floats
print(hex(id(my_int_1)), hex(id(my_int_2)))
print()

print(my_float_1 is my_float_2)
print(hex(id(my_float_1)), hex(id(my_float_2)))
```

```
True
0x7ffb39a3e388 0x7ffb39a3e388

False
0x1af9614ac30 0x1af95f48a30
```

In [28]:
```
4 != 2  # unequal
```

Out[28]:
```
True
```

Logical Operators

In [29]:
```
not True  # inversion with 'not' (NOT)
```

Out[29]:
```
False
```

In [30]:
```
# combination with 'and' - both conditions must be fulfilled (AND):

print(True and True)
print(False and True)
print(False and False)
```

```
True
False
False
```

In [31]:
```
# exclusion with 'or' - one or both conditions must be met (OR):

print(True or True)
print(False or True)
print(False or False)
```

```
True
True
False
```

In [32]:
```python
# exclusion with '^' - exactly one condition must be fulfilled (XOR):

print(True ^ True)
print(False ^ True)
print(False ^ False)
```

```
False
True
False
```

- booleans are a subclass of integers, `True` is equivalent to `1` and `False` equivalent to `0`
- therefore, `True` and `False` can be calculated with like integers

In [33]:
```python
print(int(True))
print(int(False))
```

```
1
0
```

In [34]:
```python
print(True + True)
print(True * 10)
```

```
2
10
```

- with typecasting, emtpy objects (like strings) and `None` result in `False`, all others in `True`

In [35]:
```python
print(bool(""))  # empty string
print(bool([]))  # empty list
print(bool(()))  # empty tuple
print(bool({}))  # empty dict / set
print(bool(None))
```

```
False
False
False
False
False
```

In [36]:
```python
print(bool("Moin"))
```

```
True
```

Data Types 1: Strings

String formatting & concatenation

- can be nested by alternate usage of single quotes `"` and `'`
- can be concatenated using different methods. f-strings (with leading `f"`) are the most elegant method, so please don't use the `format` and `%` methods!

In [37]:
```
print("This is a string using " + str(3) + " and " + str(4) + " as numbers.")  # using +-signs
print()
```
```
This is a string using 3 and 4 as numbers.
```

In [38]:
```
print("This is a string using", str(3), "and", str(4), "as numbers.")  # using commas -> adds a space in between
print()
```
```
This is a string using 3 and 4 as numbers.
```

In [39]:
```
print("This is a string using %s and %s as numbers." % (3, 4))  # using %s-Operator (strings or any object with a string representation)
print("This is a string using %d and %d as numbers." % (3, 4))  # using %d-Operator (integers)
print("This is a string using %.2f and %.3f as numbers." % (3, 4))  # using %.<no of digits>f-Operator (floating-point values)
print()
```
```
This is a string using 3 and 4 as numbers.
This is a string using 3 and 4 as numbers.
This is a string using 3.00 and 4.000 as numbers.
```

In [40]:
```
print("This is a string using {} and {} as numbers.".format(3, 4))  # using format()-function
print()
```
```
This is a string using 3 and 4 as numbers.
```

In [41]:
```
print(f"This is a string using {3} and {4} as numbers.")  # using f-string
print(f"This is a string using {3:.2f} and {4:.3g} as numbers.")  # using f-string with format

print(f"This is a string using {123:>3} and {34:>2} as numbers.")  # using f-string with right alignment
print(f"This is a string using {3:>3} and {4:>2} as numbers.")

print(f"This is a string using {3:e} and {4:e} as numbers.")  # using f-string with exponential notation

string_integer_1 = 3
string_integer_2 = 4
print(f"This is a string using {string_integer_1 = } and {string_integer_2 = } as numbers.")  # f-string with debugging feature
```
```
This is a string using 3 and 4 as numbers.
This is a string using 3.00 and 4 as numbers.
This is a string using 123 and 34 as numbers.
This is a string using   3 and  4 as numbers.
This is a string using 3.000000e+00 and 4.000000e+00 as numbers.
This is a string using string_integer_1 = 3 and string_integer_2 = 4 as numbers.
```

- docstrings, enclosed in triple quotes `"""` and `'''`, allow for multi-line strings

In [42]:
```
print("""This is a
multi-line
string""")
```
```
This is a
multi-line
string
```

- special characters "\n" for linebreak and "\t" for tab → "\n" allows building multi-line strings in single-quotes
- the `repr()` methods shows us these special characters
- alternatively, you can provide a *raw string* using the r-notation `r"<WHATEVER>"` (similar to-fstrings) to print the special characters

In [43]:
```
string_with_special_chars = "This is a message with a tab \t to demonstrate special characters. \n That's it."

print(string_with_special_chars)
repr(string_with_special_chars)
```

```
This is a message with a tab 	 to demonstrate special characters. 
 That's it.
```

Out[43]:
```
'"This is a message with a tab \\t to demonstrate special characters. \\n That\'s it."'
```

Manipulation of existing strings

- special characters and whitespaces at the beginning and end of strings can be removed using the `<string>.strip()` function

In [44]:
```
string_with_whitespaces = "\n This is a string with leading and trailing whitespaces. \t "

print(repr(string_with_whitespaces))
print(repr(string_with_whitespaces.strip()))
print(repr(string_with_whitespaces.rstrip()))  # remove only at the end
print(repr(string_with_whitespaces.lstrip()))  # remove only at the beginning
```

```
'\n This is a string with leading and trailing whitespaces. \t '
'This is a string with leading and trailing whitespaces.'
'\n This is a string with leading and trailing whitespaces.'
'This is a string with leading and trailing whitespaces. \t '
```

- converting strings between different cases is possible using the `<string>.upper()` and `<string>.lower()` function
- capitalise a word using the `<string>.capitalize()` function

In [45]:
```
print("HeLlO!".upper())
print("HeLlO!".lower())
print("HeLlO!".capitalize())
```

```
HELLO!
hello!
Hello!
```

character (sequence) instances in a string can be

- counted using the `<string>.count()` function → adds up the number of times they appear in the string
- found using the `<string>.find()` function → returns the position number of the first character of the sequence in the string
- replaced using the `<string>.replace()` function

In [46]:
```
print("Hellooo!".count("l"))
print("Hellooo!".find("e"))
```

```python
print("Hellooo!".replace("o", "ouu"))
```

```
2
1
Hellouuouuouu!
```

- other string-methods can be found here

Decision structures

The execution of statements can be linked to conditions:

- If the condition is `TRUE`, the `Then` statement block is executed
- If the condition is `FALSE`, all the `Elif` statement blocks are executed one-after-one
- Else, the `Else` block is executed

- If-else-trees can be nested
- writing `if <CONDITION> is True:` is not necessary - `if <CONDITION>:` suffices

- Python relies on indentation (whitespace at beginning of line) to define scope in code

In [47]:
```python
condition = True  # False
other_condition = True  # False

if condition:
    print("This is the 'then' block of the first tree.")

    if not condition:
        print("This is the 'then' block of the second tree.")  # this won't be executed
    elif not other_condition:
        print("This is the first (and only) 'elif' block of the second tree.")
    else:
        print("This is the 'else' block of the second tree.")

elif other_condition:
    print("This is the first (and only) 'elif' block of the first tree.")
else:
    print("This is the 'else' block of the first tree.")
```

```
This is the 'then' block of the first tree.
This is the 'else' block of the second tree.
```

Errors and Exceptions

- when you know that something may go wrong, you can catch the exception with a try-statement

- the `try` clause is executed → in case of an exception the `except` clause is executed
- the `else` clause is executed when there is no error
- in any case (so regardless of the outcome) the `finally` clause is executed in the end

- the `except` condition should always include a specific exception so that it does not catch unexpected errors

In [48]:
```python
integer = 5  # only works with "str(5)"

try:
    print("It is not possible to concatenate this string with the integer " + integer + ".")

except TypeError as e:
    print(f"Gotcha! It seems like there is a syntax error with the message '{e}'.")

else:
    print("No error found.")

finally:
    print("I'm the finally clause which is always executed.")
```

```
Gotcha! It seems like there is a syntax error with the message 'can only concatenate str (not "int") to str'.
I'm the finally clause which is always executed.
```

- exceptions are raised using the `raise` keyword and can include a message for the user
- a list of exceptions can be found in the Python Documentation

In [49]:
```python
raise Exception("Die!")
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
Cell In [49], line 1
----> 1 raise Exception("Die!")

Exception: Die!
```

Loops

- execute instruction blocks more than once – also called iterations

Common elements of all loop types:

1. Initialization of a counter variable (often `i`, then `j`, ...) → same variable can (oftentimes) only be used in non-nested loops - nested ones ofentimes require using different ones
2. counting function
3. termination condition

The for-loop

- fixed number of iterations
    - in range of numbers `<start>` and `<end>` using `range` function → starting with `<start>` value and ending with `<end>-1` (!) value
    - in *iterable objects* like strings, lists, tuples, dictionaries

In [50]:
```
print(type(range(1)))  # 'range' function returns object of type 'range' (and not list -> introduced later)
```

```
<class 'range'>
```

In [51]:
```
for i in range(0, 5):
    print(i)
print("---")

# range function also supports step sizes:
for i in range(0, 10, 2):
    print(i)
print("---")

# in the range function the start value defaults to 0 and can be omitted in this case:
for i in range(5):
    print(i)
print("---")

# range function can also run backwards using a negative step size:
for i in range(10, 5, -1):
    print(i)
```

```
0
1
2
3
4
---
0
2
4
6
8
---
0
1
2
3
4
---
10
9
8
7
6
```

In [52]:
```
for letter in "Hallelujah!":  # looping through a string
    print(letter)
```

```
H
a
l
l
e
l
u
j
a
h
!
```

The while-loop

- Loop is executed as long as the condition is true → condition can be changed inside the loop to break it
- `while True` loops run indefinitively

In [53]:
```python
n = 10

while n > 0:
    print(n)
    n -= 1
print("Lift off!")
```

```
10
9
8
7
6
5
4
3
2
1
Lift off!
```

In [54]:
```python
with open("customers.txt", encoding="utf-8") as file:
    line = file.readline()
    while line:                 # loop over each (non-empty) line (introduced later) -> empty lines return False, ending the loop
        print(line)             # print each line's entry.
        line = file.readline()  # read the next line
```

```
Petra;Wagstädt;23-06-89;15.99

Emil;Noltke;18-10-66;13.99

Max;Mustermann;01-01-99;159.66
```

break and continue statements

- a `break` statement, when used inside the loop, will terminate the loop and exit. If used inside nested loops, it will break out from the current loop.
- a `continue` statement, when used inside a loop, will stop the current execution, and the control will go back to the start of the loop.

In [55]:
```python
for i in range(10):
    print(f"{i = }")

    for j in range(10):

        if j > i:
            print("\t Let's get out of here...")
            break  # break also makes 'else'-statement redundant

        print(f"\t {j = }")

    continue  # continue with next iteration here: the code behind this statement is unreachable
    print("I am invisible.")
```

```
i = 0
        j = 0
        Let's get out of here...
i = 1
        j = 0
        j = 1
        Let's get out of here...
i = 2
        j = 0
        j = 1
        j = 2
        Let's get out of here...
i = 3
        j = 0
        j = 1
        j = 2
        j = 3
        Let's get out of here...
i = 4
        j = 0
        j = 1
        j = 2
        j = 3
        j = 4
        Let's get out of here...
i = 5
        j = 0
        j = 1
        j = 2
        j = 3
        j = 4
        j = 5
        Let's get out of here...
i = 6
        j = 0
        j = 1
        j = 2
        j = 3
        j = 4
        j = 5
        j = 6
        Let's get out of here...
i = 7
        j = 0
        j = 1
        j = 2
        j = 3
        j = 4
        j = 5
        j = 6
        j = 7
        Let's get out of here...
i = 8
        j = 0
        j = 1
        j = 2
        j = 3
        j = 4
        j = 5
        j = 6
        j = 7
        j = 8
        Let's get out of here...
i = 9
        j = 0
        j = 1
        j = 2
        j = 3
        j = 4
        j = 5
        j = 6
        j = 7
        j = 8
        j = 9
```

Data Types 2

Lists

- data structure consisting of a sequence of values called elements or items

Basic operations

- empty lists are initiated by brackets `my_list = []` or via the `my_list = list()` function

In [56]:
```python
my_list = ["Hi", 3, [1.5, "Bye"]]  # list storing a string, an integer, and another list
```

In [57]:
```python
# get the length of the list:
print(len(my_list))
print()
```

```
3
```

In [58]:
```python
# print the whole list:
print(my_list)
print()
```

```
['Hi', 3, [1.5, 'Bye']]
```

- elements can be all types of objects, and also of mixed type
- elements can be addressed by list name and their index: `<listname>[<index>]` → lists are an ordered collection of elements

- lists are zero-indexed (!) meaning that the first element is element number zero
- negative indices start from the back of the list → `<listname>[-1]` accesses the last element of the list

In [59]:
```python
# print (or access) a single list element:
print(my_list[0])
print()

hello_statement = my_list[0]
```

```
Hi
```

- lists are *mutable* → they (and their elements) can be modified after being created
- new items can be added to the list:
  - using the `<listname>.append(<item>)` function at the end of the list
  - using `<listname>.insert(<index>, <item>)` at a certain position → all elements after `<item>` are shifted to the right

- items can be deleted in different ways:
  - if the index is known: `del <listname>[<index>]` or `<listname>.pop(<index>)` → `pop` also returns the value, allowing it to be stored in a variable
  - if the item is known: `<listname>.remove(<item>)` → searches from beginning to end, removing the first entry

In [60]:
```python
# modify the list (in a Jupyter Notebook like this this will yield different results each time this cell is executed):
my_other_list = [3.141, 2.718]

my_list.append(my_other_list)  # add numbers as list element to list
my_list.extend(my_other_list)  # add numbers elementwise as single elements to list

first_deleted_element = my_list.pop(0)
print(first_deleted_element)

print(my_list)
```

```
Hi
[3, [1.5, 'Bye'], [3.141, 2.718], 3.141, 2.718]
```

- strings can be split into lists using the `<string>.split(<delimiter>)` function
- lists of strings can be joined using the `<delimiter>.join(<listname>)` function

In [61]:
```python
string_to_be_split = "Simply split this sentence into its words"

splitted_string_list = string_to_be_split.split(" ")
concatenated_string = " - ".join(splitted_string_list)

print(splitted_string_list)
print(type(splitted_string_list))

print()

print(concatenated_string)
print(type(concatenated_string))
```

```
['Simply', 'split', 'this', 'sentence', 'into', 'its', 'words']
<class 'list'>

Simply - split - this - sentence - into - its - words
<class 'str'>
```

List slicing

Sling describes taking a part of the list.

- omitting the first index, the slice starts at the beginning
- omitting the second index, the slice ends at the end

In [62]:
```
list_to_slice = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(list_to_slice[5:8])
print(list_to_slice[:8])  # everything up until (not including) index 8
print(list_to_slice[8:])  # everything from (including) index 8 on
print(list_to_slice[:])   # take whole list
```

```
[6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
[9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- a slice operator on the left side of an assignment can update multiple elements

In [63]:
```
list_to_slice = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
list_to_slice[1:3] = ['x', 'y']

print(list_to_slice)
```

```
[1, 'x', 'y', 4, 5, 6, 7, 8, 9, 10]
```

List combination

- lists can be combined by summing them up → other operations like subtraction do not work this way (even when both lists only contain numerical values)
- alternatively the `.extend()` method could be used

In [64]:
```
list_1_to_combine = [1, 2, 3]
list_2_to_combine = ["A", "B", "C"]

print(list_1_to_combine + list_2_to_combine)
```

```
[1, 2, 3, 'A', 'B', 'C']
```

Looping over lists

... the classical, multi-line way

In [65]:
```
loop_list = [12, 3, 1.5]

# looping directly over items:
for item in loop_list:
    print(item)
print()

# looping over indices:
for index in range(0, len(loop_list)):
    print(loop_list[index])
print()
```

```
12
3
1.5

12
3
1.5
```

- a loop over an empty list never runs the instructions

In [66]:
```
for i in []:
    print("This never happens")
print("Done.")
```

```
Done.
```

... using list comprehensions

- *list comprehensions* allow creating a new list from an old one (and other iterables) within one line of code
- same works for tuples, dictionaries, sets (all mentioned later)

Syntax: `new_list = [this_expression if condition else other_expression for item in iterable]`

- is theoretically also possible with nested expressions but much more difficult to decipher

In [67]:
```
loop_list = [12, 3, 1.5, 6, 9, 100]

# double items in value that are below 10, else triple them:

doubled_loop_list = [2*item if item < 10 else 3*item for item in loop_list]
print(doubled_loop_list)

# this is identical to (but much shorter than):

doubled_loop_list = []
for item in loop_list:
    if item < 10:
        doubled_loop_list.append(2*item)
    else:
        doubled_loop_list.append(3*item)

print(doubled_loop_list)
```

```
[36, 6, 3.0, 12, 18, 300]
[36, 6, 3.0, 12, 18, 300]
```

Other useful list methods

Sum

- summing up numercial values in a list using the `sum(<listname>)` function

In [68]:
```
summed_list = [2, 3.414, 5]
print(sum(summed_list))

# does only work when the list has numerical values exclusively:
summed_list.append("Hi")
print(summed_list)
print(sum(summed_list))
```

```
10.414
[2, 3.414, 5, 'Hi']
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [68], line 7
      5 summed_list.append("Hi")
      6 print(summed_list)
----> 7 print(sum(summed_list))

TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

Sorting

- sorting values in a list using the `<listname>.sort()` (returns `None`) or `sorted(<listname>)` (returns list object) funtions

In [69]:
```
# Numerical sorting:

unsorted_list = [10.8, 1, 3.414]

print(unsorted_list)
print(unsorted_list.sort())  # in-place operation: returns None but updates list as can be seen in next print
print()

# ---

unsorted_list = [10.8, 1, 3.414]

print(unsorted_list)
print(sorted(unsorted_list))  # returns list but does not update variable as can be seen in next print
print()
```

```
# ---

unsorted_list = [10.8, 1, 3.414]

print(unsorted_list)
print(sorted(unsorted_list, reverse=True))  # reversed sorting - sorting in descending order
```

```
[10.8, 1, 3.414]
None

[10.8, 1, 3.414]
[1, 3.414, 10.8]

[10.8, 1, 3.414]
[10.8, 3.414, 1]
```

In [70]:
```
# String sorting:
unsorted_list = ["C", "A", "T", "S"]

print(sorted(unsorted_list, reverse=True))  # also reverse sorting possible
```

```
['T', 'S', 'C', 'A']
```

In-Operator

- testing for values in a list using the `in` operator

In [71]:
```
5 in [1, 2, 3, 4, 5]
```

Out[71]:
```
True
```

Tuples

- are similar to lists: are also ordered and elements can be accessed by index
- difference being that they are *immutable* → can't be modified once they have been created

- allow collecting multiple values in one single variable

- an empty tuple is initiated with parentheses `my_tuple = ()` or with the `my_tuple = tuple()` function

In [72]:
```
# modifying a tuple raises an error:

my_tuple = (1, 2, 3)

my_tuple.append(4)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In [72], line 5
      1 # modifying a tuple raises an error:
      3 my_tuple = (1, 2, 3)
----> 5 my_tuple.append(4)

AttributeError: 'tuple' object has no attribute 'append'
```

In [73]:
```
my_tuple[0] = "A"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [73], line 1
----> 1 my_tuple[0] = "A"

TypeError: 'tuple' object does not support item assignment
```

In [74]:
```
del my_tuple[0]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [74], line 1
----> 1 del my_tuple[0]

TypeError: 'tuple' object doesn't support item deletion
```

In [75]:
```
# tuples with single values are initiated with a comma:

tuple_var = ("A")
real_tuple = ("A",)

print(type(tuple_var))
print(type(real_tuple))
```

```
<class 'str'>
<class 'tuple'>
```

Sets

- they are unordered
- once a set is created, it is not possible to change items, but they can be removed and new ones can be added
- their perk: they contain a collection of unique elements → duplicates are automatically deleted

- an empty set is initiated with the `my_set = set()` function (but not with braces)

In [76]:
```
my_set = {"A", "A", "B", "C"}

print(my_set)  # the second, non-unique element is removed

# items can be added and removed:

my_set.remove("B")
```

```
print(my_set)

my_set.add("D")
print(my_set)
```

```
{'C', 'B', 'A'}
{'C', 'A'}
{'C', 'D', 'A'}
```

In [77]:
```
# sets are unordered:

print(my_set[0])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [77], line 3
      1 # sets are unordered:
----> 3 print(my_set[0])

TypeError: 'set' object is not subscriptable
```

- allow for set-operations like determining intersections, unions, etc. → can be computationally slow with large sets

In [78]:
```
set_a = {1, 2, 3, 4}
set_b = {4, 5, 6, 7}

print(set_a.isdisjoint(set_b))
```

```
False
```

In [79]:
```
print(set_a.union(set_b))
```

```
{1, 2, 3, 4, 5, 6, 7}
```

Dictionaries

- provides a special kind of list that can use (almost) any type of value as an index (not just integers like in lists) → maps **key-value pairs**
- are unordered and mutable

Basic operations

- add new entries to the dictionary via square brackets, mapping the key to the value: `<dict>[<key>] = <value>`,
- look up value via square brackets or with the `<dict>.get(<key>)` method
    - providing the same key twice (with two different values) returns the second value

- an empty dictionary is initiated with braces `my_dict = {}` or with the `my_dict = dict()` function

In [80]:
```
adresses = {
    "David Smith": "Marketstreet 3",
    "Julia Horn": "Bridgeway 5",
    "Adam Mouse": "Main Lane 5"
}

# adding values:

adresses["Don Joe"] = "Princess Street 1"
print(adresses)
```

```
{'David Smith': 'Marketstreet 3', 'Julia Horn': 'Bridgeway 5', 'Adam Mouse': 'Main Lane 5', 'Don Joe': 'Princess Street 1'}
```

Retrieving values

In [81]:
```
# getting existing values:

print(adresses["Don Joe"])
print(adresses.get("Julia Horn"))
```

```
Princess Street 1
Bridgeway 5
```

- the bracket- and get-methods return different values for keys that do not exist in the dict

In [82]:
```
print(adresses["Max Hinte"])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In [82], line 1
----> 1 print(adresses["Max Hinte"])

KeyError: 'Max Hinte'
```

In [83]:
```
print(adresses.get("Max Hinte"))  # returns None by default which can be changed
print(adresses.get("Max Hinte", "No idea who that is."))
```

```
None
No idea who that is.
```

Looping over dictionaries

- looping directly over a dictionary is not possible, it has to be specified over which part of the dictionary is being looped
- sequence of retrieved values from loop is arbitrary since dictionary is not ordered

In [84]:
```
for key, value in adresses.items():  # returns key-value tuple which can be unpacked in loop
    print(f"{key} lives in {key}")
print()

for key in adresses.keys():
    print(key)
print()

for value in adresses.values():
    print(value)
```

```
David Smith lives in David Smith
Julia Horn lives in Julia Horn
Adam Mouse lives in Adam Mouse
Don Joe lives in Don Joe

David Smith
Julia Horn
Adam Mouse
Don Joe

Marketstreet 3
Bridgeway 5
Main Lane 5
Princess Street 1
```

External Files

Many more file operations can be found in this article.

Reading from other files

- the built-in Python `open("<storage path>")` function takes the storage path of a file (as a string) as parameter and returns a file object, which can be assigned to a variable `<filename>` to read the file contents
- when the file is not where Python is asked to look for it, a `FileNotFoundError` is raised → wrapping the `open("<storage path>")` function in a `try` statement can catch this case

- after using a file it better is closed again to avoid write-conflicts in the operating system using `<filename>.close()`
- alternatively, the `with` statement can be used which automatically closes the file

- having opened the file:
  - single lines can be read via the function `<file object>.readline()`
    - this function moves on one line whenever it is called
    - when there are no lines left in the file, it returns an empty string
  - the pointer can be set to a specific *n*-th caracter of the file using `<file object>.seek(n)>` or the current position of the pointer can be read using `<file object>.tell()>`
  - all remaining content after the current curser position can be obtained with the `<file object>.readlines()` command

In [85]:
```python
f = open("customers.txt", encoding="utf-8")  # fix output with proper "encoding"-parameter
line = f.readline()  # read the first line
print(line)
f.close()  # don't forget to close the file again!

# --- alternative:
with open("customers.txt", encoding="utf-8") as f:
    line = f.readline()  # no closing statement is required minimising the risk for file corruption
    print(line)

# --- catching exceptions:
```

```
try:  # if the 'with' statement is omitted the file has to be closed in the 'else' statement
    with open("customer_names.txt", encoding="utf-8") as f:
        line = f.readline()
        print(line)

except FileNotFoundError:
    print("File not found. Skipping read process...")
```

```
Petra;Wagstädt;23-06-89;15.99

Petra;Wagstädt;23-06-89;15.99

File not found. Skipping read process...
```

Writing to other files

- the file must be opened in *writing mode* using the `mode="w"` flag
- a file object can be written to using the `<filename>.write(<content string>)` function
- sometimes the encoding of the file has to be specified in the `open()` method using the `encoding=<ENCODING>` argument

In [86]:
```
to_be_saved = "Please save me!"

with open('output.txt', mode='w', encoding="utf-8") as f:  # can also be 'output.csv' an such
    f.write(f"{to_be_saved}\n")  # add line break character
    f.write("Alrighty, I'm on my way!")
```

Functions

- "takes" an argument and "returns" a result
    - functions have a *name*, *arguments* (also called *parameters*), and a *code body*
    - not all functions need arguments – can be called with empty arguments, too
    - not all functions return a value as their result – "void" functions have an effect, such as printing something

- since Python executes all statements in one script from top to bottom any function needs to be either defined or imported before it can be called
- when the end of the function code is reached, the flow of execution returns to the line in the original program where it was called

- Knowing how to access existing functions is a prerequisite to using external Python modules
    - Much given functionality comes in... functions
    - You need to know how to use a function by providing the right input and expecting the correct output
    - Opening a function's code and reading it, without changing it, can help you understand in how far this module does what you need

- Knowing how to organise your own code in functions lets you
    - re-use it in other areas of the program without having to type the commands out again

    - avoid typing the same code multiple times to educe errors
    - think systematically about the required and optional input for a piece of code and the resulting output

When writing or reading a program without exactly following the flow of execution through every function is termed a **leap of faith**. This means that we believe that there will be a function that can create the result that is needed because:

- it is already written and we do not want to check it now
- we will write it later and want to continue building the code that will use it
- we assume that Python provides some similar function and we will Google it later

Python also uses the leap of faith when running code that defines a function that relies on another function, which has not been defined yet (statements in the function are not run on definition, but only when the function is called) → missing or corrupt functions can create runtime errors

Imports

General remarks

- when calling a function from an imported module, that function runs code from outside the script, as defined inside the function, from top to bottom
- importing creates a module object which can be accessed

- modules not included in the standard Python library need to be downloaded, e.g. using `pip` ("pip installs python") via `pip install <module name>`
- in non-local Jupyter Notebooks you can write `!pip install <MODULE NAME>` into a code cell

In [87]:
```
import math
import numpy as np  # modules can be imported using an alias
```

In [88]:
```
# functions and constants included in the module can be accessed via a dot notation
print(math.pi)
print(math.prod([6, 2, 3]))
```

```
3.141592653589793
36
```

In [89]:
```
# generate normally distributed random numbers using numpy:

rng = np.random.normal(size=5)
print(rng)
```

```
[-0.6953451   0.85509493  0.22472847 -0.81896165  0.53189565]
```

In [90]:
```
# single functions can also be loaded into the main namespace, allowing them to be called without the dot notation
# also works with an alias 'as <xyz>'

from pandas import DataFrame, Series

print(DataFrame, Series)
```

```
<class 'pandas.core.frame.DataFrame'> <class 'pandas.core.series.Series'>
```

Special package *numpy*

- Using fixed-type arrays (vectors and matrices) is more efficient for large data sets
    - fixed-type arras cannot store any kind of data, as lists do (where every element is an object with a type), but needs less memory and computation
    - the numpy package includes the data structure ndarray for this purpose
- is convenient and efficient for mathematical operations
- vectorised computations are much faster than iterating over the array with a for-loop → this is also very handy when dealing with high-dimensional arrays as every new dimension would require an additional iteration

Initialisation - object creation

- arrays can be initialised from specific values
- array dimensions (*numpy*: "axes") are:
    - axis 0: rows (↓),
    - axis 1: columns (→)
    - axis 2: tubes (↗)
    - ...

In [91]:
```
import numpy as np

a = np.array([1, 2, 3])  # 1D array
b = np.array([[1.5, 2, 3], [4, 5, 6]])  # 2D array
c = np.array([[[1.5, 2, 3], [4, 5, 6]], [[3, 2, 1], [4, 5, 6]]])  # 3D array
```

- arrays can be inspected without actually looking at the specific values

In [92]:
```
print(c.shape)  # no of elements in each dimension:
print(c.ndim)  # no of dimensions
print(c.size)  # total no of elements in array
print(c.dtype)  # data type of values in array
```

```
(2, 2, 3)
3
12
float64
```

- array dimensions can be changed using the `array.reshape(<new shape>)` function - the new shape must be compatible with the original shape

In [93]:
```
x = np.array([1, 2, 3])
print(x.shape)
print(x)
print()

x = x.reshape((3,1))
print(x.shape)
print(x)
```

```
(3,)
[1 2 3]

(3, 1)
[[1]
 [2]
 [3]]
```

- a number of functions allow for the facilitated creation of arrays

In [94]:
```python
print(np.zeros((2, 4))) # an array of zeros of specified shape
print()

print(np.ones((2, 4), dtype=np.int16))  # an array of ones of specified shape and data type
print()

print(np.full((2, 2), 7))  # a constant array
print()

print(np.eye(3))  # a 3x3 identity matrix
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]]

[[1 1 1 1]
 [1 1 1 1]]

[[7 7]
 [7 7]]

[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

In [95]:
```python
# an array of evenly spaced values in half-open interval [start, stop)
print(np.arange(start=10, stop=25, step=5))  # step = <spacing between values>
print()

print(np.linspace(start=0, stop=2, num=9))  # num = <number of samples>
```

```
[10 15 20]

[0.   0.25 0.5  0.75 1.   1.25 1.5  1.75 2.  ]
```

- the random module allows working with (psuedo-)randomly generated numbers
- all functions return an array if an integer or a tuple of integers is given to the `shape` argument, otherwise they return a single value

- random generations can be fixed by setting a seed via the `np.random.seed(<seed>)` function

In [96]:
```python
# an array with random float values in half-open interval [0.0, 1.0)
print(np.random.random(size=(2, 2)))  # size = <shape>
print()
```

```
# an array with random integer values in half-open interval [low, high)
print(np.random.randint(low=1, high=11, size=5))  # low = <lowest int>, high = <one above highest int>, size = <shape>
print()

# an array with random float values from a normal distribution
print(np.random.normal(loc=1, scale=1, size=5))  # loc = <mean>, scale = <std>, size = <shape>
print()

# an array with random float values from a uniform distribution
print(np.random.uniform(low=1, high=11, size=5))  # low = <lowest float>, high = <highest float>, size = <shape>
print()

# an array with random float values from an exponential distribution
print(np.random.exponential(scale=0.25, size=5))  # scale = <scale parameter beta>
print()

# an array with random float values from a triangular distribution
print(np.random.triangular(left=1, mode=3, right=5, size=5))  # left = <lower limit>, mode = <distribution peak>, right = <upper limit>, size = <shape>
print()

# an array with random integers from a poisson distribution
print(np.random.poisson(lam=1, size=5))  # lam = <expected no of events in time interval>, size = <shape>
print()

# a random sample from a given 1-D array
print(np.random.choice(a=[True, False], size=(2, 2), p=[0.4, 0.6]))  # a = <1D array or int for np.arange(int)>, size = <shape>, p = <probability for each element of a (uniform if omit
```

```
[[0.29105076 0.38263551]
 [0.00624545 0.5887123 ]]

[2 5 3 4 9]

[ 2.02941806 -0.77574228  2.15382549 -0.30647283 -0.19887208]

[ 7.61138529 10.06518924  1.99146413  1.31325425  3.80997716]

[0.12422602 0.28886685 0.32764713 0.22971868 0.32055546]

[2.57133    2.38616947 2.54404664 4.43691542 3.4130161 ]

[1 0 2 1 1]

[[False False]
 [False  True]]
```

Arithmetic operations

- element-wise arithmetic operations

In [97]:
```
print(a + a)  # element-wise addition of two arrays of the same shape
print()

print(np.add(a, a))  # alternative for element-wise addition
print()

print(a + 10)  # element-wise addition of a constant
print()

print(a.dot(a.T))  # dot product with transposed array of a
```

```
[2 4 6]

[2 4 6]

[11 12 13]

14
```

Statistical operations

- all operations can also only be applied to certain axes

In [98]:
```
print(a.mean())
print()

print(np.mean(a))  # alternative for mean-method of array
print()

print(a.std())
print()

print(np.median(a))  # no array method available for median
print()

print(a.max())
print()

print(a.sum())
print()

print(a.cumsum())  # the cumulative sum: sum of values until element n
```

```
2.0

2.0

0.816496580927726

2.0

3

6

[1 3 6]
```

Selecting, filtering & aggregating data

- you can select data from arrays using their index
- conditions arise from element-wise comparisons

In [99]:
```
a = np.array([1, 7, 4, 12, 18, 2])

print(a < 10)  # selection condition: element value smaller than 10 -> creates array of booleans
print()

filtered_a = a[a < 10]  # select values from array based on boolean array
print(filtered_a)
print()

# this method also works with two arrays
b = np.array([6, 3, 12, 10, 9, 21])

filtered_b = b[a < 10]
print(filtered_b)
```
```
[ True  True  True False False  True]

[1 7 4 2]

[ 6  3 12 21]
```

- replace array values using numpy's `where(<condition>, <array>, <replacement value>)` method

In [100…
```
a = np.random.rand(2, 2)
condition = a > 0.5

print(a)
print()

print(condition)
print()

print(np.where(condition, a, 0))  # if condition is true, use original value from a, else use new value of 0
```
```
[[0.97894042 0.17388139]
 [0.60547381 0.20048938]]

[[ True False]
 [ True False]]

[[0.97894042 0.        ]
 [0.60547381 0.        ]]
```

- slicing in numpy is possible just like with lists using `[start:end:step]`
  - dimensions are separated using commas `,`
  - all values of a dimension are selected using a colon `:`
  - when the start index is not passed it is considered 0, when the end index is not passed it is considered the length of the array in that dimension, when the step index is not passed it is considered 1
  - the result *includes* the start index, but *excludes* the end index
- slices return views rather than copies of the data – changing them changes the original array

In [101…
```
original_array = np.random.randint(1, 100, (5, 4))

print(original_array)
print()
```

```
sliced_array = original_array[:2, 1:4]
print(sliced_array)
print()

sliced_array[0:3, 1] = -5
print(sliced_array)
```

```
[[49  8 98 29]
 [43 36  2 51]
 [81 41 99 56]
 [ 3 94 39 92]
 [62 91 67  3]]

[[ 8 98 29]
 [36  2 51]]

[[ 8 -5 29]
 [36 -5 51]]
```

- aggregations over multidimensional arrays, like `np.sum(<array>)` or `np.count_nonzero(<array>)`, require specifying an axis when only a part of the array is supposed to be aggregated

In [102…
```
a = np.random.rand(2, 2)

print(a)
print()

print(np.count_nonzero(a, axis=1))
```

```
[[0.10270728 0.50849925]
 [0.32133968 0.39128868]]

[2 2]
```

Other operations

- array values can be rounded

In [103…
```
a = np.random.rand(2, 2)

print(a)
print()

print(np.round(a, 2))
print()

print(a.round(3))
```

```
[[0.23096684 0.71258969]
 [0.69027927 0.8433491 ]]

[[0.23 0.71]
 [0.69 0.84]]

[[0.231 0.713]
 [0.69  0.843]]
```

- array values can be sorted

In [104…

```
a = np.random.rand(5)

print(a)
print()

print(np.sort(a))  # alternatively use a.sorted() for in-place sorting
```

```
[0.88768953 0.49684614 0.69019035 0.07257198 0.54973429]

[0.07257198 0.49684614 0.54973429 0.69019035 0.88768953]
```

- multidimensional arrays can be reduced in dimensionality ("flattened")

In [105…

```
a = np.array([[1,2], [3,4]])

print(a)
print(a.shape)
print()

a = a.flatten()

print(a)
print(a.shape)
```

```
[[1 2]
 [3 4]]
(2, 2)

[1 2 3 4]
(4,)
```

Special package *pandas*

- pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with “relational” or “labeled” data both easy and intuitive
- the two primary data structures of pandas are *Series* (1-dimensional) and *DataFrames* (2-dimensional)
- Dataframes can be regarded as a specialised dictionary which is a collection of multiple Series (comparable to an Excel table)
    - every column has its own title
    - every row is indexed by an index
- Dataframes rely on numpy arrays for efficient data storage
    - every column in one data frame is a series that can be referenced via the column name
    - every set of values in a series is a numpy array

Initialisation - object creation

- external data can be read from different formats, e.g. using the `pd.read_csv()` or the `pd.read_excel()` functions

- external files can be created in different formats, e.g. using the `pd.to_csv()` or the `pd.to_excel()` functions

In [106…
```
import numpy as np
import pandas as pd

df = pd.read_csv("customers.txt", header=None, delimiter=";", names=["first_name", "last_name", "birthday", "value_of_bought_items"])

print(df)
```

```
  first_name   last_name  birthday  value_of_bought_items
0      Petra    Wagstädt  23-06-89                  15.99
1       Emil      Noltke  18-10-66                  13.99
2        Max  Mustermann  01-01-99                 159.66
```

- from existing data inside the python script one can create a Series by passing a list of values, letting pandas create a default *integer index*

In [107…
```
s = pd.Series([1, 3, 5, np.nan, 6, 8])
print(s)
```

```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

- creating a DataFrame is possible by passing a NumPy array -- here with a *datetime index* using `date_range()` and labeled columns
- a DataFrame can have more than one index

In [108…
```
dates = pd.date_range("20230101", periods=6)
print(dates)
print()

df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list("ABCD"))
print(df)
print()
```

```
DatetimeIndex(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04',
               '2023-01-05', '2023-01-06'],
              dtype='datetime64[ns]', freq='D')

                   A         B         C         D
2023-01-01 -1.426046 -0.320458  1.468820 -0.157254
2023-01-02 -0.467924 -0.312949 -1.301537 -0.633109
2023-01-03 -0.477372 -0.382931 -0.219437 -0.808027
2023-01-04 -0.931789 -1.210188 -0.879141  2.089094
2023-01-05  0.249563 -2.091596  0.103610  0.220959
2023-01-06 -0.371545 -1.547860  0.654393 -1.535336
```

- creating a DataFrame is also possible by passing a dictionary of objects that can be converted into a series-like structure

In [109…
```
df2 = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20130102"),
        "C": pd.Series(1, index=list(range(4)), dtype="float32"),
        "D": np.array([3] * 4, dtype="int32"),
        "E": pd.Categorical(["test", "fail", "test", "train"]),
        "F": "foo",
    }
)
print(df2)
```

```
     A          B    C  D      E    F
0  1.0 2013-01-02  1.0  3   test  foo
1  1.0 2013-01-02  1.0  3   fail  foo
2  1.0 2013-01-02  1.0  3   test  foo
3  1.0 2013-01-02  1.0  3  train  foo
```

- the columns of the resulting DataFrame have different data types

In [110…
```
print(df2.dtypes)
```

```
A          float64
B    datetime64[ns]
C          float32
D            int32
E         category
F           object
dtype: object
```

- use `DataFrame.head()` and `DataFrame.tail()` to view the top and bottom rows of the frame respectively

In [111…
```
print(df.head(3))  # standard value is first five rows
```

```
                   A         B         C         D
2023-01-01 -1.426046 -0.320458  1.468820 -0.157254
2023-01-02 -0.467924 -0.312949 -1.301537 -0.633109
2023-01-03 -0.477372 -0.382931 -0.219437 -0.808027
```

- display the index or columns of the DataFrame

In [112…
```
print(df.index)
print()
print(df.columns)
```

```
DatetimeIndex(['2023-01-01', '2023-01-02', '2023-01-03', '2023-01-04',
               '2023-01-05', '2023-01-06'],
              dtype='datetime64[ns]', freq='D')

Index(['A', 'B', 'C', 'D'], dtype='object')
```

- you can sort a DataFrame by index or by column values

Sorting, selecting & updating data

- data can be sorted based on colum or index values

In [113…
```
df = df.sort_index(axis=1, ascending=False)
print(df)
print()

df = df.sort_values(by=["B","C"])  # first sort by column B, then by C
print(df)
```

```
                   D         C         B         A
2023-01-01 -0.157254  1.468820 -0.320458 -1.426046
2023-01-02 -0.633109 -1.301537 -0.312949 -0.467924
2023-01-03 -0.808027 -0.219437 -0.382931 -0.477372
2023-01-04  2.089094 -0.879141 -1.210188 -0.931789
2023-01-05  0.220959  0.103610 -2.091596  0.249563
2023-01-06 -1.535336  0.654393 -1.547860 -0.371545

                   D         C         B         A
2023-01-05  0.220959  0.103610 -2.091596  0.249563
2023-01-06 -1.535336  0.654393 -1.547860 -0.371545
2023-01-04  2.089094 -0.879141 -1.210188 -0.931789
2023-01-03 -0.808027 -0.219437 -0.382931 -0.477372
2023-01-01 -0.157254  1.468820 -0.320458 -1.426046
2023-01-02 -0.633109 -1.301537 -0.312949 -0.467924
```

- data can be filtered

In [114…
```
print(df["A"])  # select single column -> returns Series
print()

print(df[0:3])  # select rows by index number
```

```
2023-01-05    0.249563
2023-01-06   -0.371545
2023-01-04   -0.931789
2023-01-03   -0.477372
2023-01-01   -1.426046
2023-01-02   -0.467924
Name: A, dtype: float64

                   D         C         B         A
2023-01-05  0.220959  0.103610 -2.091596  0.249563
2023-01-06 -1.535336  0.654393 -1.547860 -0.371545
2023-01-04  2.089094 -0.879141 -1.210188 -0.931789
```

In [115…
```
# get DataFrame subset ...
print(df.loc["20230102":"20230104", ["A", "B"]])  # ... by index values (inclusive) and column names
print()

print(df.iloc[3:5, 0:2])  # ... by numeric value positions
```

```
                   A         B
2023-01-04 -0.931789 -1.210188
2023-01-03 -0.477372 -0.382931
2023-01-02 -0.467924 -0.312949

                   D         C
2023-01-03 -0.808027 -0.219437
2023-01-01 -0.157254  1.468820
```

In [116…

```python
# get single value from DataFrame ...
print(df.at["20230102", "A"])  # ... by index value and column name
print()

print(df.iat[1, 1])  # ... by numeric value position
```

```
-0.4679236630135978

0.6543927344713328
```

In [117…

```python
# boolean indexing
condition_1 = df["A"] > 0

print(condition_1)  # boolean Series
print()

print(df[condition_1])  # filtered DataFrame
print()

print(df[condition_1 & (df["B"] < -1)])  # multiple conditions connected with logical operators -> enclose in parentheses
print()

print(df2[df2["E"].isin(["test", "fail"])])  # special functions are available for boolean indexing
```

```
2023-01-05     True
2023-01-06    False
2023-01-04    False
2023-01-03    False
2023-01-01    False
2023-01-02    False
Name: A, dtype: bool

                   D        C         B         A
2023-01-05  0.220959  0.10361 -2.091596  0.249563

                   D        C         B         A
2023-01-05  0.220959  0.10361 -2.091596  0.249563

     A          B    C  D     E    F
0  1.0 2013-01-02  1.0  3  test  foo
1  1.0 2013-01-02  1.0  3  fail  foo
2  1.0 2013-01-02  1.0  3  test  foo
```

- values can also be updated

In [118…

```python
print(df)
print()
```

```
df.iat[0, 1] = 0
df.loc[:, "D"] = np.array([5] * len(df))

print(df)
```

```
                   D         C         B         A
2023-01-05  0.220959  0.103610 -2.091596  0.249563
2023-01-06 -1.535336  0.654393 -1.547860 -0.371545
2023-01-04  2.089094 -0.879141 -1.210188 -0.931789
2023-01-03 -0.808027 -0.219437 -0.382931 -0.477372
2023-01-01 -0.157254  1.468820 -0.320458 -1.426046
2023-01-02 -0.633109 -1.301537 -0.312949 -0.467924

            D         C         B         A
2023-01-05  5  0.000000 -2.091596  0.249563
2023-01-06  5  0.654393 -1.547860 -0.371545
2023-01-04  5 -0.879141 -1.210188 -0.931789
2023-01-03  5 -0.219437 -0.382931 -0.477372
2023-01-01  5  1.468820 -0.320458 -1.426046
2023-01-02  5 -1.301537 -0.312949 -0.467924
```

```
C:\Users\Niclas Grocholski\AppData\Local\Temp\ipykernel_19864\1865755326.py:5: FutureWarning: In a future version, `df.iloc[:, i] = newvals` will attempt to set the values inplace inst
ead of always setting a new array. To retain the old behavior, use either `df[df.columns[i]] = newvals` or, if columns are non-unique, `df.isetitem(i, newvals)`
  df.loc[:, "D"] = np.array([5] * len(df))
```

- whole columns or rows can be deleted

In [119…

```
del df["D"]
df = df.drop("C", axis=1)  # {0 or ‘index’, 1 or ‘columns’} with respective column or index names

print(df)
```

```
                   B         A
2023-01-05 -2.091596  0.249563
2023-01-06 -1.547860 -0.371545
2023-01-04 -1.210188 -0.931789
2023-01-03 -0.382931 -0.477372
2023-01-01 -0.320458 -1.426046
2023-01-02 -0.312949 -0.467924
```

- missing values can be filled using the `<Series/DataFrame>.fillna(<value>)` method

In [120…

```
s = pd.Series([1, 2, 3, float("nan"), 5, np.nan])

print(s)
print()

s = s.fillna(10)

print(s)
```

```
0    1.0
1    2.0
2    3.0
3    NaN
4    5.0
5    NaN
dtype: float64

0     1.0
1     2.0
2     3.0
3    10.0
4     5.0
5    10.0
dtype: float64
```

Statistics

- pandas offers a variety of statistics functions

In [121…
```
print(df.describe())  # overview function
print()

print(df["A"].mean())  # example of a statistics function
```

```
              B         A
count  6.000000  6.000000
mean  -0.977664 -0.570852
std    0.754653  0.564583
min   -2.091596 -1.426046
25%   -1.463442 -0.818185
50%   -0.796560 -0.472648
75%   -0.336077 -0.395639
max   -0.312949  0.249563

-0.5708518953130498
```

Applying functions to values individually

- `DataFrame.apply()` applies a user defined function to the data

In [122…
```
print(df.apply(lambda x: x.max() - x.min()))
```

```
B    1.778647
A    1.675609
dtype: float64
```

String manipulation & string selection

- *string methods* allow for manipulations and selections of string values

In [123…
```
s = pd.Series(["A", "B", "C", "Aaba", "Baca", np.nan, "CABA", "dog", "cat"])

print(s.str.lower())
print()

print(s[s.str.contains("a", na=False)])
```

```
0       a
1       b
2       c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object

3    Aaba
4    Baca
8     cat
dtype: object
```

Merging data

- with SQL-like merging actions multiple dataframes can be combined

In [124…

```
df3 = pd.DataFrame(np.random.randn(10, 4))

# concatenating pandas objects together along an axis
print(pd.concat([df3[:1], df3[3:7], df3[9:]]))
```

```
          0         1         2         3
0 -1.398749  0.307527  1.214785  0.348959
3  1.809976  0.155614 -0.495836  0.862537
4  0.034344 -0.516362 -1.471037 -0.616612
5  0.243768  0.908830  0.476220  0.806703
6 -0.105163  0.705397 -0.493831 -0.519046
9 -0.196222  0.411308  0.588949 -0.865907
```

In [125…

```
left = pd.DataFrame(
    {
        "key": "foo",
        "lval": [1, 2]
    }
)
right = pd.DataFrame(
    {
        "key": "foo",
        "rval": [4, 5]
    }
)

print(left)
print()

print(right)
print()
print(pd.merge(left, right, on="key"))  # inner join
```

```
   key  lval
0  foo     1
1  foo     2

   key  rval
0  foo     4
1  foo     5

   key  lval  rval
0  foo     1     4
1  foo     1     5
2  foo     2     4
3  foo     2     5
```

Grouping data

- By *Grouping* we are referring to a process involving one or more of the following steps:
  - *Splitting* the data into groups based on some criteria
  - *Applying* a function to each group independently
  - *Combining* the results into a data structure

In [126…

```
df4 = pd.DataFrame(
    {
        "A": ["foo", "bar", "foo", "bar", "foo", "bar", "foo", "foo"],
        "B": ["one", "one", "two", "three", "two", "two", "one", "three"],
        "C": np.random.randn(8),
        "D": np.random.randn(8),
    }
)

print(df4)
print()

print(df4.groupby(["A", "B"]).sum())
```

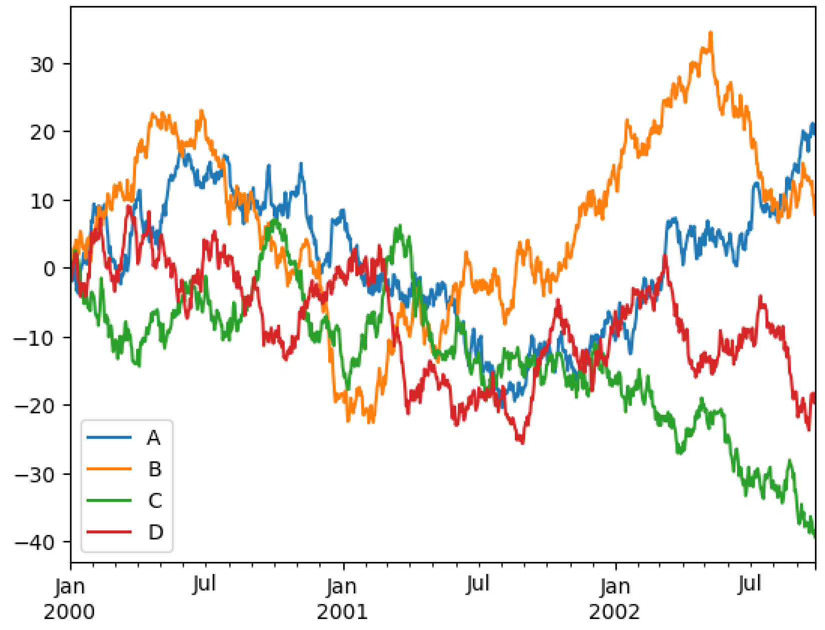
```
     A      B         C         D
0  foo    one -0.647955  0.409460
1  bar    one -1.294978 -1.000920
2  foo    two -0.464644  1.296563
3  bar  three -0.904877 -0.298006
4  foo    two -0.886103 -1.699937
5  bar    two -1.843789 -2.221722
6  foo    one  0.269556  1.398924
7  foo  three  1.754985 -0.757826

                  C         D
A   B                        
bar one   -1.294978 -1.000920
    three -0.904877 -0.298006
    two   -1.843789 -2.221722
foo one   -0.378399  1.808384
    three  1.754985 -0.757826
    two   -1.350748 -0.403374
```

Generating plots

- pandas can use matplotlib to plot data from DataFrames

In [127…

```
import matplotlib.pyplot as plt

df5 = pd.DataFrame(
    np.random.randn(1000, 4),
    index=pd.date_range("1/1/2000", periods=1000),
    columns=["A", "B", "C", "D"]
)

df5 = df5.cumsum()

df5.plot()
plt.legend(loc='best')

plt.show()
plt.close()
```

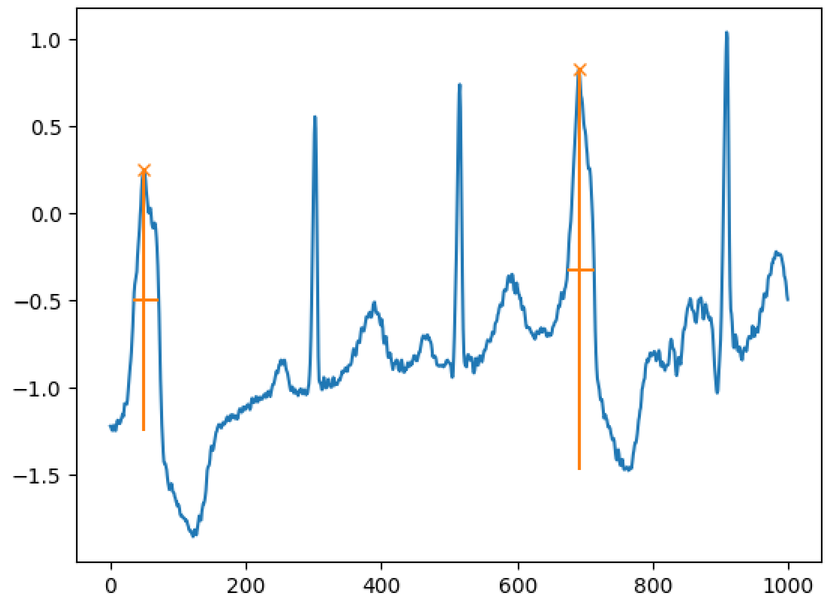


Special package *scipy*

- scipy includes algorithms for optimization, integration, interpolation, eigenvalue problems, algebraic equations, differential equations and many other classes of problems
- scipy has too many functions to show here so take this example as a teaser what the library may be used for

In [128…

```
from scipy.signal import find_peaks
from scipy.datasets import electrocardiogram
import matplotlib.pyplot as plt

# select the atypical heart beats from an electrocardiogram based on minimum peak widths and prominences

x = electrocardiogram()[17_000:18_000]
peaks, properties = find_peaks(x, prominence=1, width=20)

plt.plot(x)
plt.plot(peaks, x[peaks], "x")
plt.vlines(x=peaks, ymin=x[peaks] - properties["prominences"],
           ymax = x[peaks], color = "C1")
plt.hlines(y=properties["width_heights"], xmin=properties["left_ips"],
           xmax=properties["right_ips"], color = "C1")

plt.show()
plt.close()
```



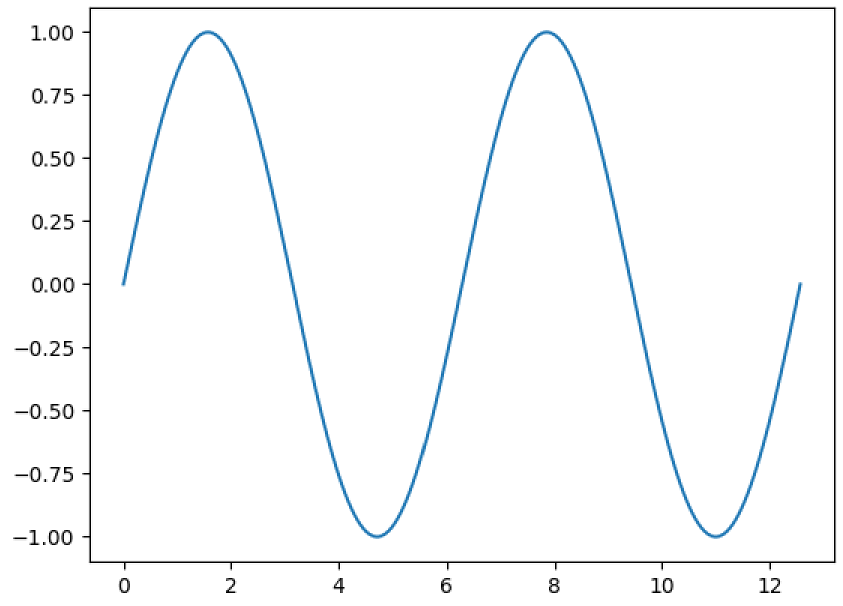
Special package *matplotlib*

Types of plots

- matplotlib is a library for making 2D plots
- it is designed with the philosophy that you should be able to create simple plots with just a few commands

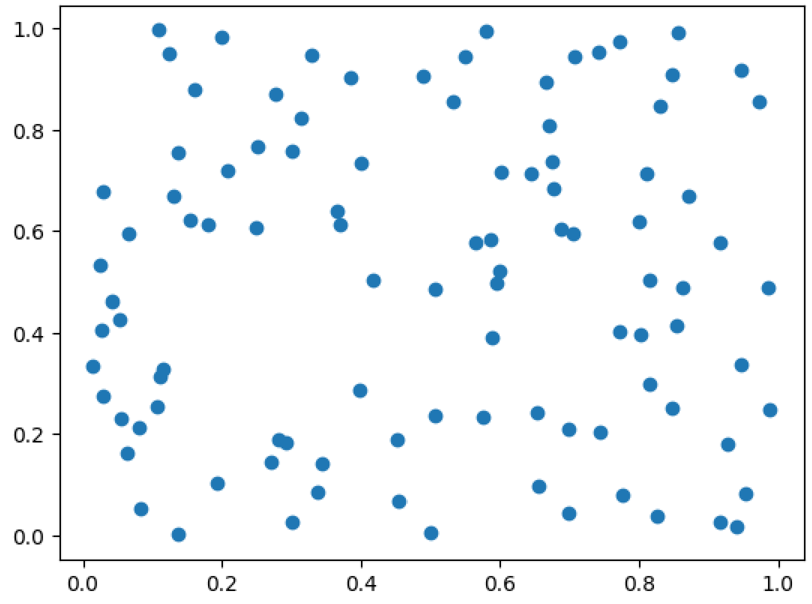
In [129…

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

# prepare
x = np.linspace(0, 4 * np.pi, 1000)
y = np.sin(x)

# render
plt.plot(x, y)

# observe
plt.show()
plt.close()
```



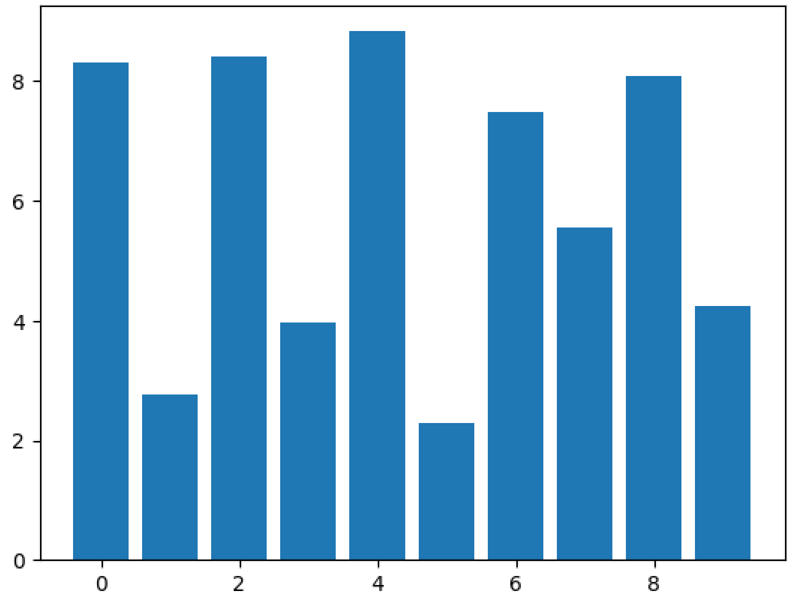
- matplotlib offers several kinds of plots

In [130…
```
x = np.random.uniform(0, 1, 100)
y = np.random.uniform(0, 1, 100)

plt.scatter(x, y)

plt.show()
plt.close()
```



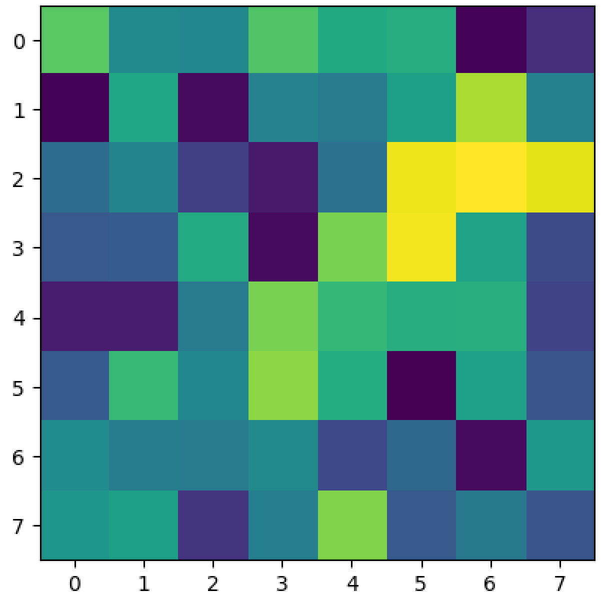
In [131…

```
x = np.arange(10)
y = np.random.uniform(1, 10, 10)

plt.bar(x, y)

plt.show()
plt.close()
```



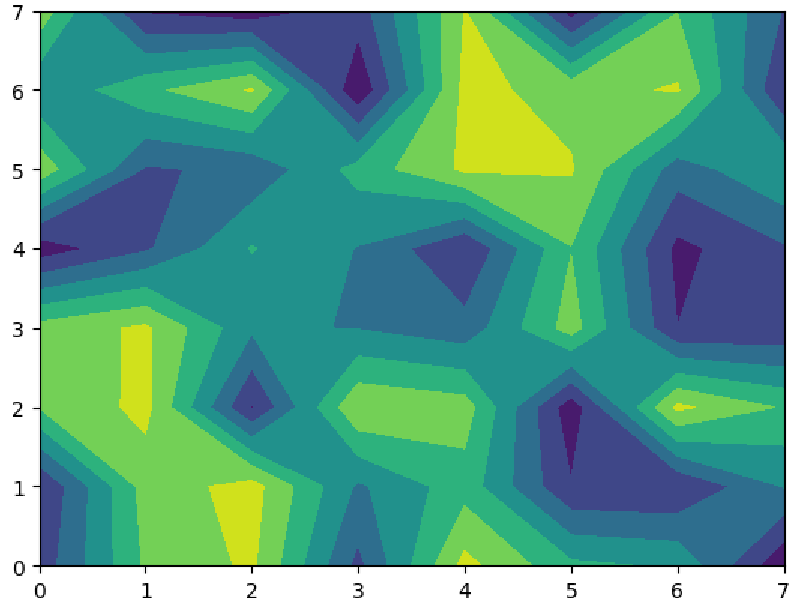
In [132…

```
z = np.random.uniform(0, 1, (8, 8))

plt.imshow(z)

plt.show()
plt.close()
```



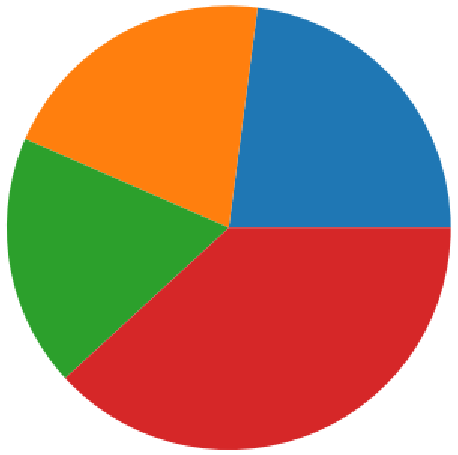
In [133…

```
z = np.random.uniform(0, 1, (8,8))

plt.contourf(z)

plt.show()
plt.close()
```



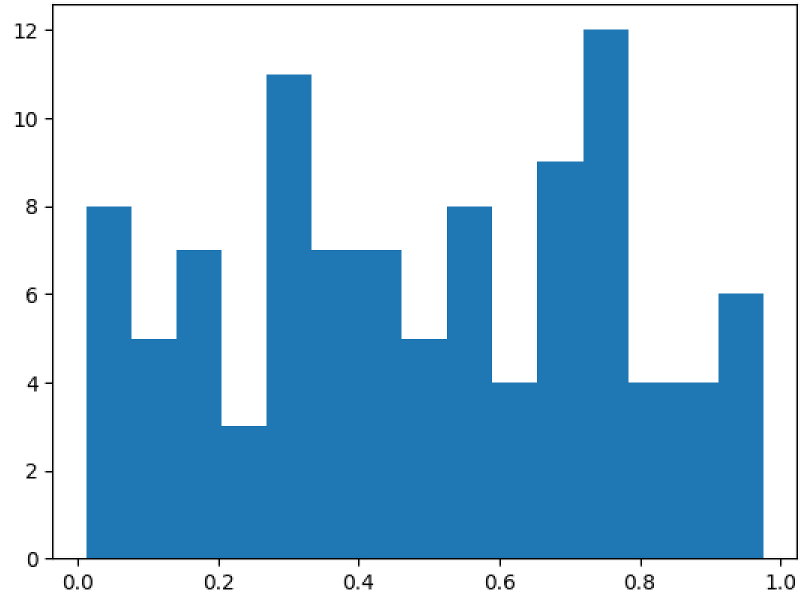
In [134…

```
z = np.random.uniform(0, 1, 4)

plt.pie(z)

plt.show()
plt.close()
```



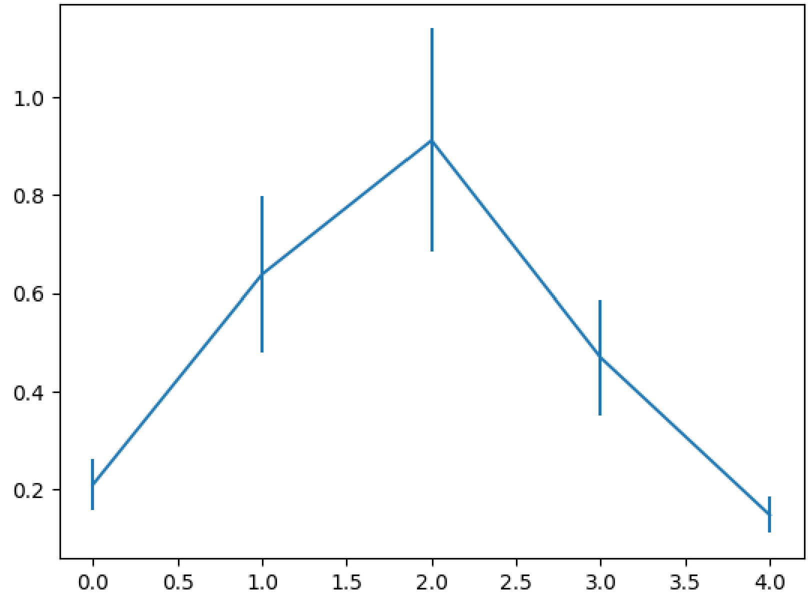
In [135…

```
z = np.random.uniform(0, 1, 100)

plt.hist(z, bins=15)

plt.show()
plt.close()
```



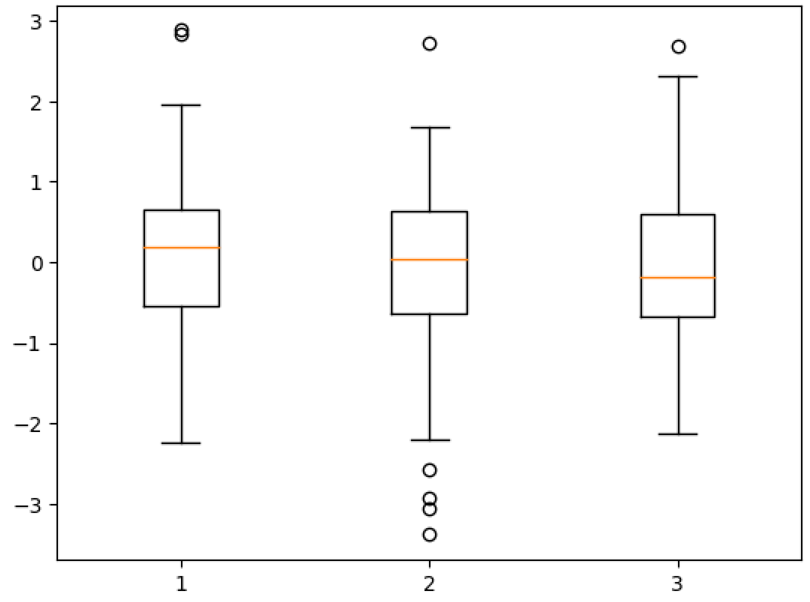
In [136…

```
x = np.arange(5)
y = np.random.uniform(0, 1, 5)

plt.errorbar(x, y, y/4)

plt.show()
plt.close()
```



In [137…

```
z = np.random.normal(0, 1, (100,3))

plt.boxplot(z)

plt.show()
plt.close()
```

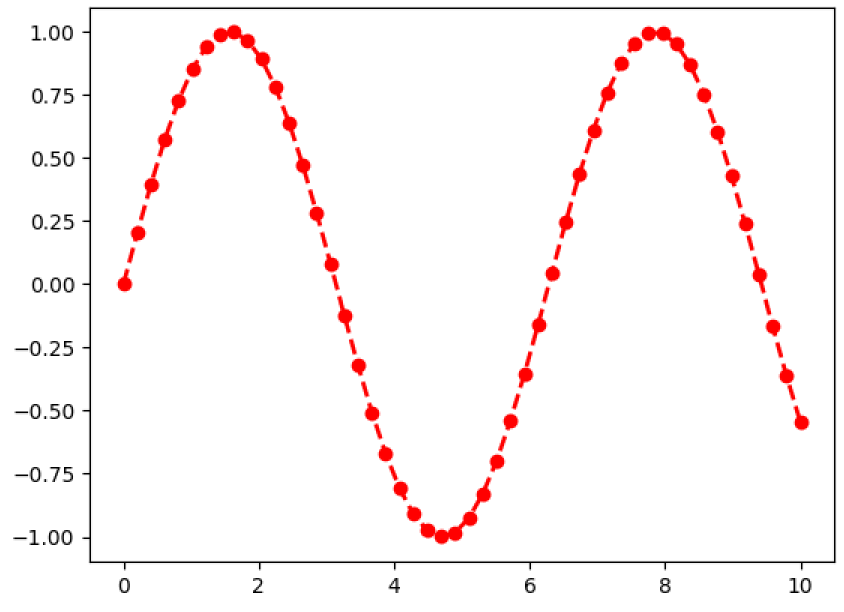


Make plots prettier

- you can modify pretty much anything in a plot, including limits, colors, markers, line width and styles, ticks and ticks labels, titles, etc.

In [138…

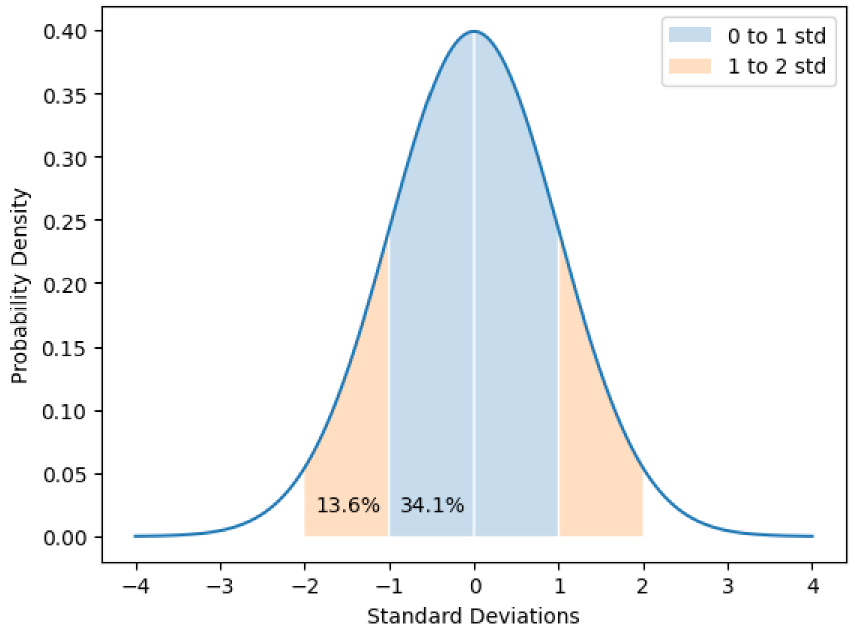
```
x = np.linspace(0, 10, 50)
y = np.sin(x)

plt.plot(x, y, color="red", linestyle="--", linewidth=2, marker="o")

plt.show()
plt.close()
```



- areas of the plot can be filled with colors

In [139…

```
import matplotlib.pyplot as plt
import numpy as np
from collections import namedtuple

x = np.linspace(-4,4, 1000)  # sample 1000 values between -4 and 4
y = (1 / (np.sqrt(2 * np.pi))) * np.exp(-0.5 * x**2)  # formula for the normal distribution's pdf

plt.plot(x, y)

plt.fill_between(x, y,
                 where=((x <= -0.01) & (x >= -0.99)) | ((x >= 0.01) & (x <= 0.99)),
                 alpha=0.25, label="0 to 1 std")
plt.text(-0.475, 0.025, "34.1%", ha="center", va="center")

plt.fill_between(x, y,
                 where=((x <= -1.01) & (x >= -2)) | ((x >= 1.01) & (x <= 2)),
                 alpha=0.25, label="1 to 2 std")
plt.text(-1.475, 0.025, "13.6%", ha="center", va="center")


plt.legend()

plt.xlabel("Standard Deviations")
plt.ylabel("Probability Density")

plt.show()
plt.close()
```

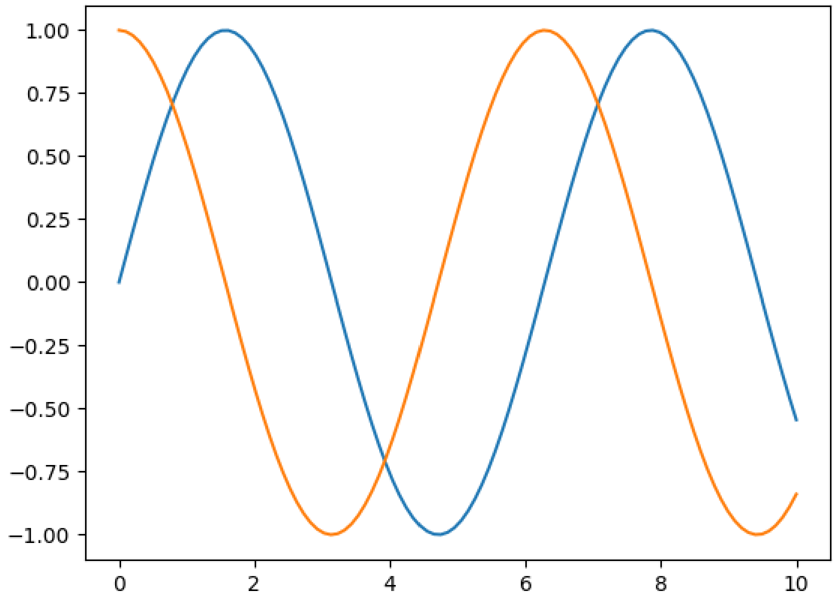


Plot multiple data

- You can plot several data on the the same figure, but you can also split a figure in several subplots (named *axes*):

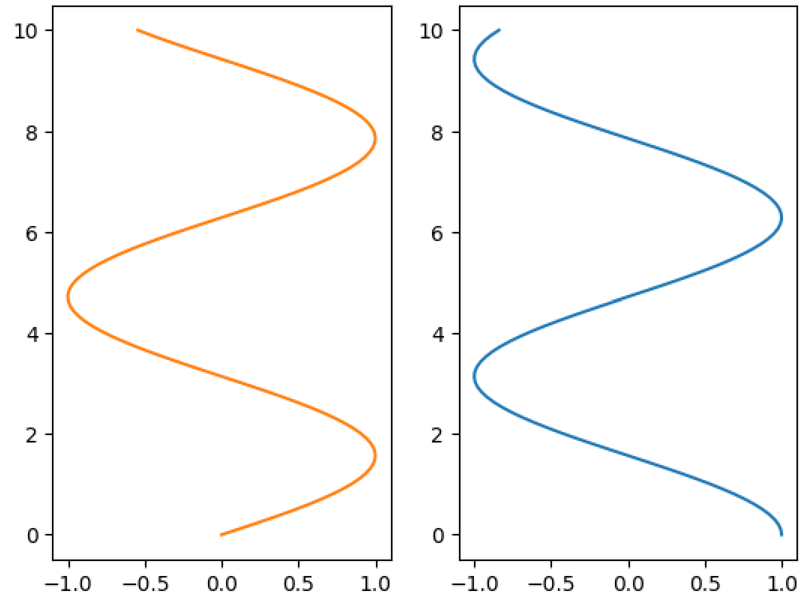
In [140…

```
x = np.linspace(0, 10, 100)
y1, y2 = np.sin(x), np.cos(x)

plt.plot(x, y1)
plt.plot(x, y2)

plt.show()
plt.close()
```



In [141…

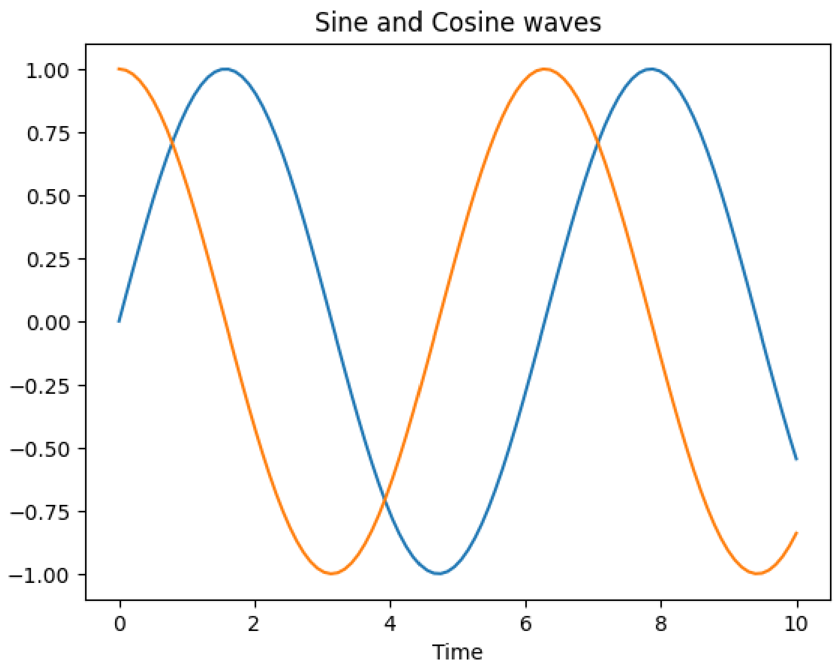
```
x = np.linspace(0, 10, 100)
y1, y2 = np.sin(x), np.cos(x)

fig, (ax1, ax2) = plt.subplots(1,2)

ax1.plot(y1, x, color="C1")
ax2.plot(y2, x, color="C0")

plt.show()
plt.close()
```



- you can annotate a plot with labels

In [142…

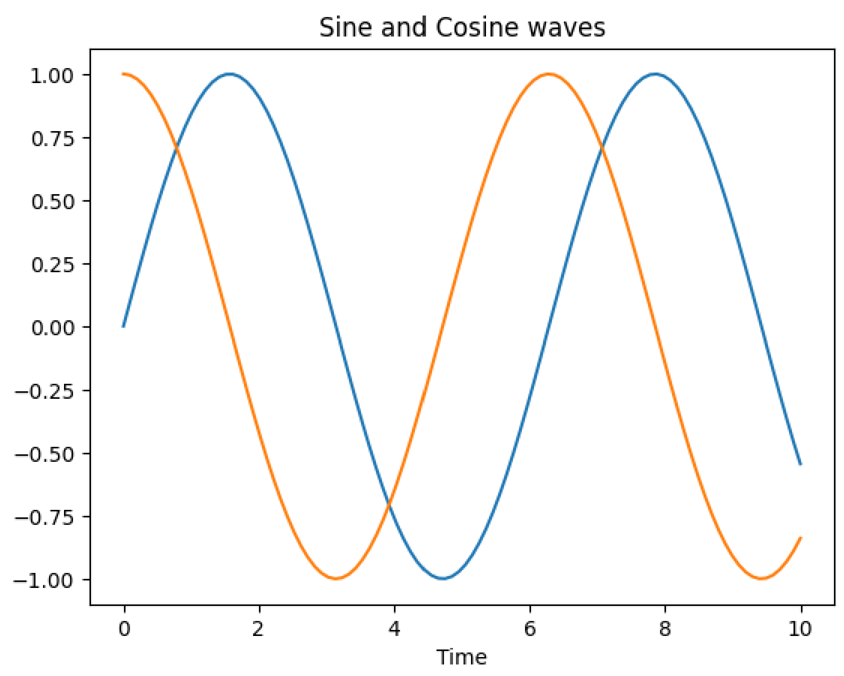
```python
x = np.linspace(0, 10, 100)
y1, y2 = np.sin(x), np.cos(x)

fig, ax = plt.subplots()

ax.plot(x, y1)
ax.plot(x, y2)

ax.set_title("Sine and Cosine waves")

ax.set_ylabel(None)
ax.set_xlabel("Time")

plt.show()
plt.close()
```



- matplotlib allows you to export figures as bitmap or vector images

In [143…

```
x = np.linspace(0, 10, 100)
y1, y2 = np.sin(x), np.cos(x)

fig, ax = plt.subplots()

ax.plot(x, y1)
ax.plot(x, y2)

ax.set_title("Sine and Cosine waves")

ax.set_ylabel(None)
ax.set_xlabel("Time")

fig.savefig("fig.png", dpi=300)
fig.savefig("fig.pdf", bbox_inches="tight")
```



Defining own Functions

General remarks

Being able to create ones own building blocks is one of the key tenants of programming: write a function once, use it and use it again whenever and wherever you need it.

- functions are defined using the `def` statement

```
def function_name(argument_name_1, argument_name_2 = default_value_2, ...):
    """docstring"""

    <code body>
```

- *Docstrings* are optional and describe what the function does. There are different styles for docstrings that are currently used, for example Google has defined their own style

In [144…
```
# a function is an object, which has the type function
print(type(np.random.normal))
```

```
<class 'builtin_function_or_method'>
```

- functions define variables locally meaning that they can't be accessed outside the function
- inside the function variables from outside the function can, however, be accessed
- functions can be nested which leads to nested name spaces

In [145…
```
def local_variable_function():
    local_variable_function_variable = 5

local_variable_function_variable
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [145], line 4
      1 def local_variable_function():
      2     local_variable_function_variable = 5
----> 4 local_variable_function_variable

NameError: name 'local_variable_function_variable' is not defined
```

In [146…
```
global_variable = 3

def outer_function():
    outer_function_variable = 12

    def inner_function():
        inner_function_variable = 6

        print(global_variable * outer_function_variable * inner_function_variable)

    inner_function()
    print(global_variable * outer_function_variable * inner_function_variable)  # this throws an error

outer_function()
```

```
216
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [146], line 14
     11     inner_function()
     12     print(global_variable * outer_function_variable * inner_function_variable)  # this throws an error
---> 14 outer_function()

Cell In [146], line 12, in outer_function()
      9     print(global_variable * outer_function_variable * inner_function_variable)
     11 inner_function()
---> 12 print(global_variable * outer_function_variable * inner_function_variable)

NameError: name 'inner_function_variable' is not defined
```

- one function can also call another function, even if that function is defined afterwards

In [147…
```
def function_1():
    print("Foo")

def function_2():
    function_1()
    print("Faa")

function_2()

# ---
```

```
def function_3():
    function_4()  # only defined later -> still works since all functions are loaded into memory before execution of the script
    print("Fii")

def function_4():
    print("Fee")

function_3()
```

```
Foo
Faa
Fee
Fii
```

- functions can recursively call themselves
- recursions are not exactly efficient and Python sets a limit of about 1000 calls that a function can make to itself

In [148…
```
def recursive_countdown(n):
    if n <= 0:
        print("Lift off!")
    else:
        print(n)
        recursive_countdown(n-1)

recursive_countdown(5)
```

```
5
4
3
2
1
Lift off!
```

Arguments

- A function can take any number and type of variables as argument (or also called parameter) → what the function needs has to be described when defining it
- the type of the argument does not (but can) be defined when defining the function

In [149…
```
def function_name(argument_1, argument_2):
    pass  # don't do anything ("placeholder" keyword)
```

In [150…
```
# calling a function that requires an argument without providing one leads to a TypeError

def call_me_maybe(my_name):
    print(f"I'll maybe call you, {my_name}.")

call_me_maybe()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [150], line 6
      3 def call_me_maybe(my_name):
      4     print(f"I'll maybe call you, {my_name}.")
----> 6 call_me_maybe()

TypeError: call_me_maybe() missing 1 required positional argument: 'my_name'
```

- by defining a default value for the argument when setting up the function, an argument can be made optional (or "default")
- when the function is called without values for optional arguments then the defaults are assumed, otherwise the values are overwritten
- optional arguments all have to be listed behind the mandatory arguments in the function definition

In [151…

```python
def compute_random_numbers(number_of_numbers = 5):
    print(np.random.rand(number_of_numbers))

compute_random_numbers()  # 5 numbers are created by default
compute_random_numbers(3)  # but 3 is also possible
```

```
[0.15898941 0.05177788 0.27958234 0.14622756 0.5836053 ]
[0.88502537 0.40661722 0.54939353]
```

- by providing *argument keywords* when calling a function, it is possible to call a function without having to stick to the order of arguments given in the definition
- adding a `*;` in the function definition forces the use of keywords in all following parameters

In [152…

```python
def interest(capital, rate=0.1):
    print(f"Interest is {capital*rate}")

interest(100)  # omitting default argument value
interest(200, 0.4)  # specifiying both without argument names -> values have to be in order as defined in function
interest(rate=0.5, capital=500)  # providing argument names allows switching the sequence of arguments

def interest_with_keywords(*, capital, rate=0.1):
    print(f"Interest with keywords is {capital*rate}")

interest_with_keywords(capital=500, rate=0.5)
interest_with_keywords(200, 0.4)
```

```
Interest is 10.0
Interest is 80.0
Interest is 250.0
Interest with keywords is 250.0
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [152], line 12
      9     print(f"Interest with keywords is {capital*rate}")
     11 interest_with_keywords(capital=500, rate=0.5)
---> 12 interest_with_keywords(200, 0.4)

TypeError: interest_with_keywords() takes 0 positional arguments but 2 were given
```

- functions are defined once and every time the function is called its current state is updated → mutable arguments will lead to unexpected function behaviour → *do not* use mutable default arguments in Python

In [153…
```
def mutable_arguments_are_forbidden(mutable_argument = []):
    mutable_argument.append("some stuff")
    print(f"A list based on {', '.join(mutable_argument)}.")

mutable_arguments_are_forbidden()
mutable_arguments_are_forbidden()  # the last state of the argument 'mutable_argument' is preserved
mutable_arguments_are_forbidden()
```

```
A list based on some stuff.
A list based on some stuff, some stuff.
A list based on some stuff, some stuff, some stuff.
```

- as Python does not define the type of an argument when defining the function (as in "this function wants two numeric values") the type of the argument has to be checked inside the function
- the type of a variable can be checked using the `isinstance(<variable>, <type>)` function or by catching an exception

In [154…
```
def multiplication(factor_1, factor_2):
    print(factor_1 + factor_2)

multiplication(factor_1="100", factor_2=50)  # providing a string triggers an exception
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [154], line 4
      1 def multiplication(factor_1, factor_2):
      2     print(factor_1 + factor_2)
----> 4 multiplication(factor_1="100", factor_2=50)  # providing a string triggers an exception

Cell In [154], line 2, in multiplication(factor_1, factor_2)
      1 def multiplication(factor_1, factor_2):
----> 2     print(factor_1 + factor_2)

TypeError: can only concatenate str (not "int") to str
```

In [155…
```
def multiplication(factor_1, factor_2):
    if isinstance(factor_1, (int, float)) and isinstance(factor_2, (int, float)):
        print(factor_1 * factor_2)
    else:
        print("You can only multiply two numbers.")

multiplication(factor_1="100", factor_2=50)  # if it was not for the test, multiplying the string with the number would concatenate the string "100" 50 times
multiplication(factor_1=100, factor_2=50)
```

```
You can only multiply two numbers.
5000
```

Functions with and without returns

- so far, the functions had an effect by printing some statements, but is was not possible to take the result out of the function
- to write a function that returns a value (which can then be assigned to a variable), include a `return` statement
- none of the statements after the `return` statement are executed!

In [156…
```
def double_number(n):
    return 2*n
    print("I'm invisible!")
```

```
doubled_number = double_number(2)
print(doubled_number)
```

```
4
```

- generator functions look and act just like regular functions, but with one defining characteristic: they use the `yield` keyword instead of `return`
- `yield` indicates where a value is sent back to the caller, but unlike `return`, you don't exit the function afterward → instead, the state of the function is remembered
- generators are a great way to optimize memory as they do not store all their contents in memory when they are being called the first time

In [157...

```
gen_1 = (num**2 for num in range(4))
print(type(gen_1))

for i in gen_1:
    print(i)
```

```
<class 'generator'>
0
1
4
9
```

- instead of using a `for` loop, you can also call `next()` on the generator object directly to get the next output from the generator

In [158...

```
def sequence():
    num = 0
    while num < 3:
        yield num
        num += 1

gen_2 = sequence()

print(next(gen_2))
print(next(gen_2))
print(next(gen_2))
```

```
0
1
2
```

- generators throw a `StopIteration`-Error when they run out of values to create
- `next` functions can define a default value to return in this case

In [159...

```
print(next(gen_2))
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In [159], line 1
----> 1 print(next(gen_2))

StopIteration:
```

Storing a set of functions or parameters in a module

- any python script `<module_name>.py` can be considered as a module
- to store functions or parameters in a module that can be re-used
  - Define them in a separate script file, which should be stored where you also store the script you are currently working with
  - Import the module via the import statement referencing the script name
- access imported functions or parameters as previously discussed, e.g. `<module_name>.do_something()`
- all functions in the imported module are defined in the course of the import statement – this may trigger errors if the function definitions are flawed!
- more on absolute and relative imports can be read in PEP 328

- use this functionality to split the code of your project into different files to prevent single files from becoming overly cluttered (more on this here)

In [160…
```
import my_functions  # import functions from file "my_functions.py" which is located in the same folder as the executed script

my_functions.hello()
```
```
Hello MyFunctions!
```

Selected Built-in functions

Python isn't heavily influenced by functional languages but by imperative ones. However, it provides several features that allow you to use a functional style

Anonymus Functions: lambda

- an anonymous function is a function without a name
- `lambda` functions are regularly used with the built-in functions `map()` and `filter()` as well as in the key parameters of the `sorted()`, `min()`, and `max()` functions

- anonymous functions are created with the `lambda` keyword, one or more bound variables (`x, y`), and a body (`x + y`)

In [161…
```
lambda x, y: x + y
```
Out[161]:
```
<function __main__.<lambda>(x, y)>
```

- the function can be immediately applied to an argument by surrounding the function and its argument with parentheses

In [162…
```
(lambda x, y: x + y)(2, 3)
```
Out[162]:
```
5
```

- the function can be assigned to a variable to be called later, though this is considered bad practice

In [163…
```
foo = lambda x: x + 1
foo(3)
```
Out[163]:
```
4
```

Map

- `map(<function>, <iterable>, [<iterable2>, ...])` applies a function to each element of a list

In [164…

```
# map using a lambda-function
l = list(map(lambda x: x**2, [1, 2, 3, 4, 5]))  # map returns a map-object which needs to be converted to a list
print(l)
print()


# map using a self-defined function
def add(i):
    return i + i

l = list(map(add, [1, 2, 3, 4, 5]))
print(l)
print()


# map using a built-in function
l = list(map(abs, [-1, 2, -3, 4, -5]))
print(l)
print()


# map using 2 lists and a function with 2 inputs
def square_and_add(i, j):
    return i ** 2 + j

l = list(map(square_and_add, [-1, 2, -3, 4, -5], [6, 7, 8, 9, 10]))
print(l)
print()
```

```
[1, 4, 9, 16, 25]

[2, 4, 6, 8, 10]

[1, 2, 3, 4, 5]

[7, 11, 17, 25, 35]
```

Filter

- `filter(<function>, <iterable>)` applies a function to each element of a list and returns only those elements for which the function returns `True`

In [165…

```
def above_100(n):
    return n > 100

l = list(filter(above_100, [1, 2, 100, 108, 12_603, 12]))
print(l)
print()
```

```
[108, 12603]
```

- the package `itertools`, has a function called `filterfalse()` that does the inverse of `filter()`

Zip

- the `zip(<iterable 1>, <iterable 2>, ...)` function takes iterables, aggregates elements of the same positions in a tuple, and returns a zip (iterator) object

In [166…

```
coordinates = ['x', 'y', 'z']
values = [3, 4, 5]

z = zip(coordinates, values)

print(z)
print()

print(list(z))  # cast to list to view elements of iterator
```

```
<zip object at 0x000001AFBC0BDB00>

[('x', 3), ('y', 4), ('z', 5)]
```

- the shortest iterable determines the length of the iterator → later elements of longer iterables are omitted

In [167…

```
two_value_list = [1, 2]
four_value_list = [3, 4, 5, 6]

print(list(zip(two_value_list, four_value_list)))  # only 2 tuples in output list since shorter input list only had two elements
```

```
[(1, 3), (2, 4)]
```

- the zip function is especially useful when iterating over multiple iterables of connected values since the tuple elements can be unpacked in the loop

In [168…

```
for coordinate, value in zip(coordinates, values):
    print(f"Coordinate {coordinate} has a value of {value}")
```

```
Coordinate x has a value of 3
Coordinate y has a value of 4
Coordinate z has a value of 5
```

Classes

General remarks

Python supports:

- *imperative programming* – by implementing algorithms as step-by-step changes in a single state (flow of execution from top to bottom with small detours for functions)
- *procedural programming* – by letting the user define functions
- *functional programming* – particularly via a module termed `functools` (not covered here)

- *object-oriented programming* – letting the user define **classes** as templates for objects

Object oriented programming aggregates data and functions into **classes of realworld objects**. Classes organise objects that:

- have the same *attributes*
- can use the same *methods*

They combine the best of two worlds:

- storing data that relates to one thing in one place
- making a set of functionality as relevant to one thing available together
- thing = object

Class objects are initiated using the `class` statement

```
class ClassName:
    """Optional docstring describing class"""

    def __init__(self, attribute_1, attribute_2 = default_value_2, ...):
        <code block>

    def method_1(self):
        <code block>
```

Classes are factories for objects:

- Creating a new object is called instantiation: any object is an instance of a class – one variant of parametrising the class template
- To create an object (an *instance* of a class), call the class as if it was a function, e.g.: `new_instance = ClassName()`

Naming conventions:

- the names of classes are writen in "camel case": are capitalised and written as one word (e.g. "CamelCase")
- the names of variables and objects are written in "snake case": with lower case letters separated by underscores (e.g. "snake_case")

Class names are obtained using the `type()` function

In [169…

```
# class creation:
class Point:
    """Represents a point in 2D-space."""

# instance creation:
new_point = Point()

# attribute definition and value assignment:
new_point.x_coordinate = 15.5
new_point.y_coordinate = 66.5
```

```
# retrieving attribute values:
print(f"The point is at x={new_point.x_coordinate} and y={new_point.y_coordinate}")
```

```
The point is at x=15.5 and y=66.5
```

- it is not necessary to pre-define attributes in the class definition to store them in an object
- attemting to access an attribute that was not defined before will result in an `AttributeError`

In [170…

```
new_point_2 = Point()

# the attributes 'x_coordinate' and 'y_coordinate' were only defined for the instance 'new_point' of the class 'Point'
# but not for the instance 'new_point_2'
print(f"The point is at x={new_point_2.x_coordinate} and y={new_point_2.y_coordinate}")
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In [170], line 5
      1 new_point_2 = Point()
      3 # the attributes 'x_coordinate' and 'y_coordinate' were only defined for the instance 'new_point' of the class 'Point'
      4 # but not for the instance 'new_point_2'
----> 5 print(f"The point is at x={new_point_2.x_coordinate} and y={new_point_2.y_coordinate}")

AttributeError: 'Point' object has no attribute 'x_coordinate'
```

As any new class is defined as a type like any other (integer, float, ...), when assigning a value to an object, it is also possible to assign a new object (an instance of another class) to the existing object.

- by simply assigning an object to a new variable, it is NOT copied but only aliased: the two variables will still refer to the same instance → this applies to all objects and can be annoying when overlooked
- this can be avoided by using the `copy()` function of the `copy` module

In [171…

```
class Article:
    """"""

class Customer:
    """"""

banana = Article()
banana.price = 5
banana.comments = ["Yellow", "Fruity"]

new_customer = Customer()
new_customer.favourite_article = banana


# testing the identity of the attributes of the two classes:
print(hex(id(new_customer.favourite_article.price)), hex(id(banana.price)))
print(new_customer.favourite_article.price is banana.price)

# testing the effect of an alteration of the attribute of one class to the other:
banana.price = 6
print(new_customer.favourite_article.price)

banana.comments.append("Soft")
print(new_customer.favourite_article.comments)
```

```
0x7ffb39a3e3a8 0x7ffb39a3e3a8
True
6
['Yellow', 'Fruity', 'Soft']
```

In [172…

```python
# aliasing only applies to more complex types, not to integers, floats, strings:
int_1 = 5
int_2 = int_1

print(int_2)
int_1 = 6
print(int_2)
print()

# aliasing does also apply to lists and dictionaries!
list_1 = [1, 2, 3]
list_2 = list_1

print(list_2 is list_1)
list_1.append(4)
print(list_1)
print(list_2)
print()

# copying prevents accidental changes:
from copy import copy

list_2 = copy(list_1)

print(list_2 is list_1)
list_1.append(5)
print(list_1)
print(list_2)
```

```
5
5

True
[1, 2, 3, 4]
[1, 2, 3, 4]

False
[1, 2, 3, 4, 5]
[1, 2, 3, 4]
```

Modifier functions

When being taking (pointers to) objects as arguments, functions can also work as modifiers, adjusting values within the object.

- This can be efficient, but can also make programs harder to track and debug.
- Anything that can be done in this way can also be achieved by handling return statements.

Methods: class functions

- by defining functions as part of the object, so-called *methods*, the function can access all information that is already there
- any object instantiated from a class with that method (and only those) has access to that method

The `__init__` method:

- by implementing an init method and relying on it in the further code, one can avoid building inconsistent instances of a class and making mistakes in attribute assignment
- this predefined type of method gets automatically invoked whenever an instance is instantiated

When a class has an `__init__` method defined, objects can be instantiated from it

- without arguments, so that attributes get assigned the standard values
- with arguments, so that attributes get assigned the values handed to the method

The `__str__` method:

- invoked when the instance is printed
- by defining this method for a specific class, it can be defined what is printed when an object from that class is printed

In [173…

```python
import math


class Circle:
    def __init__(self, diameter):
        self.diameter = diameter  # by the provided diameter value 'diameter' is bound to a local 'self.diameter' object variable
        self.area = None  # not absolutely necessary to invoke attribute as None before calculation function is called, but recommended

    def __str__(self):
        return f"This is a circle with the diameter {self.diameter}."

    def calculate_area(self):  # calculation could also be done directly in __init__
        self.area = (math.pi/4) * self.diameter ** 2


small_circle = Circle(3)

print(small_circle.area)
small_circle.calculate_area()  # Note: methods are called with opening and closing parentheses...
print(small_circle.area)       #       ... and attributes without those!
```

```
None
7.0685834705770345
```

Useful class-operations

Decorators: Dataclasses and Properties

- a decorator function (`@decorator`) is a function that adds new functionality to an existing function - the decorator function is wrapped around the existing function
- `dataclasses` provides a decorator and functions for automatically adding generated special methods such as `__init__()` and `__repr__()` to user-defined classes

- the `property` decorators allows using a class method like a class attribute; the value is updates every time the method is called

In [174…
```python
from dataclasses import dataclass, field

@dataclass
class Item:
    value: int


@dataclass
class Knapsack:
    capacity: int
    items: list[Item] = field(default_factory=lambda: [])  # alternative to initialising an empty list: field(default_factory=lambda: list)

    @property
    def is_full(self) -> bool:
        return len(self.items) >= self.capacity

    def add_item(self, item: Item) -> None:
        self.items.append(item)


k = Knapsack(capacity=10)
print(k)
print(k.is_full)
```

```
Knapsack(capacity=10, items=[])
False
```

Listing and checking attributes and methods

- with the `vars()` function a dictionary of all attributes and their respective values is returned
- with the `dir()` function a list of all methods is returned

In [175…
```python
print(vars(small_circle))
print()

print(dir(small_circle))
```

```
{'diameter': 3, 'area': 7.0685834705770345}

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass_
_', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'area',
'calculate_area', 'diameter']
```

- the `hasattr(<object>, <attribute name>)` method returns a boolean values that indicates if an object has an attribute of a certain name

In [176…
```python
print(hasattr(small_circle, "area"))
print(hasattr(small_circle, "size"))
```

```
True
False
```

Sorting lists of class instances by attribute values

- the sorting functions have a `key` parameter which can be used in combination with a `lambda` function to sort items based on their attribute values

In [177…

```python
import numpy as np

l = [Item(value=np.random.randint(1, 30)) for _ in range(5)]

print(l)
print()

l = sorted(l, key=lambda item: item.value, reverse=True)

print(l)
```

```
[Item(value=13), Item(value=14), Item(value=9), Item(value=22), Item(value=26)]

[Item(value=26), Item(value=22), Item(value=14), Item(value=13), Item(value=9)]
```

Discrete Event Simulation using SimPy

Overview

- modeled system is dynamic over time and characterized by its state
- over time, events occur at particular time points and change the system state
- The simulation control processes events in the order of their occurrence and thereby moves through simulated time → the simulated time does not pass smoothly

Entity (*simpy*: component):

- *permanent*: stays in the system (*simpy*: resource, e.g. machine)
- *temporary*: moves through the system (*simpy*: process, e.g. product)

Entity queue: list of temporary entities with status waiting for an interaction

- usually assigned to resources
- processing can be FIFO (first in, first out), LIFO (last in, first out), random, ...
- processing may depend on entity characteristics (priority queue)

Entity state:

- *busy*: state during activity
- *idle*: state before and after activity

Events: milestones in a discrete system

- describe all system state changes - between events, the system state remains constant
- does not use up time - takes place at a single point in time

Event list: list of all events with their respective time of occurrence

Simulation clock: variable stating the current time in the simulation model

Statistics indicator: variable storing statistic information about the system

SimPy Implementation

In [178…
```
import simpy
```

Environment

- active components (like vehicles, customers or messages) are modeled with *processes*
- all processes live in an *environment*, the environment instance has to be passed to all processes and resources

```
env = simpy.Environment()  # set up environment
```

- the simulation time is defined in the run command `env.run(until = <time>)`
- the simulation time is unitless but can be interpreted to have a unit (e.g. hours, minutes)

```
env.run(until = 100)
```

Processes

- processes interact with the environment and with each other via *events*
- when a process yields an event, the process gets suspended - the process is resumed (triggered) when the event occurs

- processes are described by Python generators
- during their lifetime, they create events and `yield` them in order to wait for them to be triggered
- the event time can be tracked using `env.now` (returns a numerical value)

- *timeout events* `env.timeout(<time>)` are triggered after a certain amount of (simulated) time has passed
- they allow a process to sleep (hold its state) for the given time

```
def the_process(env, start_time, index):
    yield env.timeout(start_time)  # the process starts with a time delay

    print(f"Process {index} starting at {env.now}")
    yield env.timeout(5)  # the process has a duration of 5
    print(f"Process {index} ending at {env.now}")
```

- processes have to be added to the simulation:

```
env.process(the_process(env, 0, 1))
```

- When a process is interrupted a `simpy.Interrupt` exception is thrown
- the exception can be caught, defining what needs to be done in case of an interruption

Resources

(Classical) Resources:

- with classical resources (*Resources*), requests are handled first come first serve
- requests that arrive while a resource is busy are added to a queue, the process can then only continue when the resource is available again

- a resource can deal with as many events at the same time as its `capacity` attribute allows
- the `count` attribute returns the number of occupied resource units (hence `<resource>.capacity - <resource>.count` is number of free resource units)

```
classical_resource = simpy.Resource(capacity=1)  # capacity must be positive integer (defaults to 1)


def resource_process_alternative_1(env: simpy.Environment, resource: simpy.Resource, start_time: int, index: int):
    yield env.timeout(start_time)

    res = resource.request()  # requesting uses 1 unit of the resource
    yield res  # wait for access: only once 1 unit of the resource is available, the process can resume

    print(f"Process {index} got resource at {env.now} ({resource.capacity - resource.count} of {resource.capacity} resource units are available from now on) - starting process")

    yield env.timeout(5)
    print(f"Process {index} ending at {env.now} ({resource.capacity - resource.count} of {resource.capacity} resource units are available from now on)")

    resource.release(res)  # the resource has to be released manually



def resource_process_alternative_2(env: simpy.Environment, resource: simpy.Resource, start_time: int, index: int):
    yield env.timeout(start_time)

    with resource.request() as res:  # the request is automatically released when the request was created within a with statement
        yield res  # either way, the request has to be yielded

        print(f"Process {index} got resource at {env.now} ({resource.capacity - resource.count} of {resource.capacity} resource units are available from now on)  - starting process")

        yield env.timeout(5)
        print(f"Process {index} ending at {env.now} ({resource.capacity - resource.count} of {resource.capacity} resource units are available from now on)")
```

Priority Resources:

- *Priority Resources* work similarly to Resources but consider the priority of a request
- more important requests will gain access to the resource earlier than less important ones
- priority is expressed by integer numbers (also negative) in the `priority` attribute of the request → smaller numbers mean a higher priority
- this way, although a process requested the resource later than another one, it could use it earlier because its priority was higher

```
classical_resource = simpy.Resource(capacity=1)


def priority_resource_process_alternative_1(env: simpy.Environment, resource: simpy.Resource, start_time: int, index: int, prio: int):
    ...  # same as before

    res = resource.request(priority=prio)
    yield res

    ...  # same as before

    resource.release(res)  # the resource has to be released manually
```

Preemtive Resources:

- *Preemptive Resources* work similarly to PriorityResources but can also be interrupted by very important requests
- unlike classical and priority resources, preemtive resources use a special

```
res = simpy.PreemptiveResource(env, capacity=1)  # capacity must be positive integer (defaults to 1)
```

- the *Preemptive Resource* adds a preempt flag (that defaults to `True`) to `<resource>.request()`
- a process can decide not to preempt another resource user by setting the flag to `False`
- the implementation of *Preemptive Resources* values priorities higher than preemption → preempt requests are not allowed to cheat and jump over a higher prioritised request

```
def preemt_resource_process_alternative_1(env: simpy.Environment, resource: simpy.Resource, start_time: int, index: int, prio: int, preemt: bool):
    ...  # same as before

    res = resource.request(priority=prio, preemt=preemt)
    print(f"Process {index} requesting at {env.now}")

    try:
        yield req
        print(f"Process {index} got resource at {env.now} ({resource.capacity - resource.count} of {resource.capacity} resource units are available from now on) - 
starting process")

        yield env.timeout(5)
        print(f"Process {index} ending normally at {env.now} ({resource.capacity - resource.count} of {resource.capacity} resource units are available from now on)")

    except simpy.Interrupt:
        print(f'Process {index} got preempted at {env.now}')

    finally:
        resource.release(res)  # the resource has to be released manually
```

Coding Concepts

Development approach: Prototype and Patch

1. First, write code that runs without errors but does not need to quite deliver the result you want it to deliver in the end
2. In later steps of development, you add code to “patch” the outcome from the first prototype, so that it achieves what you want
3. In a final stage, you can clean up the code so that it achieves the desired results efficiently.

- consider listing individual steps as comments (everything after a `#` in a code block), so you don’t “forget” anything → consider using the `TODO` keyword
- write the first lines and see whether the program is working - it does not need to do everything you want it to do yet!
- add and change the existing code incrementally - use variables to hold intermediary values to be able to keep track of them
- Test after each incremental step

Advantages:

- testing after incremental steps shows where an error is likely to come from
- writing code incrementally reduces large and complex tasks into manageable steps

Disadvantages:

- after you have written a working program incrementally, consider reworking it to avoid unnecessary steps or to improve computational efficiency

Testing and Debugging

System validation

- aims to ensure system reliability, recognise flaws in design or code, and to correct them.
- Professional projects allocate the same amount of time to testing as to implementing (50-50-rule)
- Difficult decision: When to stop testing?
    - Only the most creative (destructive) personnel should be testing
    - Programmers should never test their own code
    - Test cases should be developed by well-trained users during or after the definition of requirements
    - Testing can show the presence of bugs, but never show their absence (E.W. Dijkstra)

Formal Structure

1. *Test Case Identifier*: can be a number or a unique name
2. *Test Case Description*: so that whoever does the testing knows what the test is for
3. *Test Data*: needs to allow for efficient testing but also include special cases
4. *Expected Result*: comparing expected and actual outcomes

Test-Driven Design

1. Write the test of the desired functionality.
2. Run it and confirm that it fails as expected.

3. Then amend the code so that the test will succeed.

- Do nothing until you have written a test for it.
- Relies on functional, black-box, end-to-end, acceptance tests: What the user expects to see.
- Relies on automated testing for efficiency.

Types of Bugs

- *Specification* – wrong model: e.g., incorrect formula
- *Omission* – model misses something: e.g., did not take learning into account
- *Data* – faulty parameters: e.g., mis-estimated willingness to pay
- *Syntactical* – wrong format of statement: e.g., miss „;" at the end of a command in Java
- *Logical* – wrong conditioning: e.g., created an endless loop

Types of Errors

Syntax error:

- problems in the structure of a program (comparable to spelling and grammatical errors in writing)
- shows up as soon as trying to run the program when the Python interpreter encounters code that it cannot understand
- E.g.: `number = 8+2)`

Runtime error:

- code is syntactically correct but tries things that are not allowed or defined
- also called exception
- error does not appear until after the program has started running
- can be caught by Try-Catch blocks

Semantic error:

- related to meaning/logic of the code
- program runs without generating error messages but does not compute the right values
- does not do "the right thing"
- can be tricky to find – needs meaningful test cases and plausibility checks

Errors when a function is not working

- Something is wrong with the arguments the function receives
- Something is wrong within the function
- Something is wrong with the return value or the way it is being used

→ Insert print statements in relevant areas of code
→ Insert checks (if statements) to ensure valid values

Errors from copying and aliasing

- recognise aliased structures by printing its memory adress `print(hex(id(<var>)))`
- When you want to adjust a list (by removing items or sorting it), it can make sense to store a copy to not lose the original information (using the `copy()` function)

Errors when reading and writing files

- can result from whitespace, as spaces, tabs and newlines are normally invisible
- the function `repr(<var>)` can help, as it represents whitespace characters with the backslash sequences that introduces them in strings

Avoiding errors when using objects

- use `isinstance(<obj>, <class>)` to check whether objectname is an instance of classname or to check for the data type stored in a variable
- `hasattr(<obj>, <attribute name>)` checks whether objectname has an attribute termed attributename

Helpful information to be found in error messages

Where did the error occur?

- Which script file?
- Which module?
- Which line?

What type of error was it?

→ read them carefully but do not assume that they tell you everything or that everything they tell you is correct

Tips on Debugging

1. **Find ways to duplicate** the bug
    - Find a parameter setting where the bug occurs every time
    - Fix random seeds to avoid variation
2. **Describe** the bug
    - The act of stating the problem often brings its source to surface
3. Always assume it is a bug **in your code**
    - It is not a marvellous new finding
    - It is not a problem with the language or the framework
4. **Divide and conquer**
    - Find out which parts of the program are not buggy
    - Build the simplest model and setting that still duplicates the error

5. **Be creative**
    - Try out alternative ways of doing what you want to do, not to solve, but to eliminate the bug
6. **Leverage tools**
    - Use the environment’s step-by-step features, variable monitors, stop points etc.
7. Take notes and **go step by step**
    - Keep track of variable values
    - Close the doors and shut out every distraction-
    - If it helps, do it in pairs
8. **Verify** that the bug is fixed
    - Try out whether you find new settings to bring the bug back

Things to try when debugging

- *Reading*: read the code back to yourself
- *Running*: make experiments with different versions of the code
- *Ruminating*: What kind of error? What is information from the error message? What did you do before the error started to appear?
- *Rubberducking*: Explain the problem to someone else – or to a rubber duck.
- *Retreating*: Undo recent changes until the code is working again and then start rebuilding

- use small test cases to avoid long run times
- write checks that summarise the data, e.g. instead of printing out the list, try checking for the number of items to see whether it is complete
- Write checks in the code
    - ”sanity check”: check for values that should never occur
    - ”consistency check”: compute the same result through two ways and check whether the results actually match
- Format your print statements to let you quickly see if something is wrong