Python for Computational Science

January 19, 2025

Outline

Outline

Outline

Outline

Python for Computational Science

Computational Science

- use of computers to support research and operation in science, engineering, industry and services
- applications include
    - analysis of data and visualisation
    - data science / data analytics
    - artificial intelligence (AI) & machine learning (ML)
    - control
    - computer simulations
    - virtual design & optimisation
    - symbolic mathematics

Computational Science - What are objectives?

Minimum objective

- solve the given (science/data) problem using computation

Ideally also

- test the software
- document and archive the software
- make the study reproducible
- make the study re-usable
- make the software re-usable
- be time-efficient in developing the software, and
- be time-efficient in executing the software

Computational Science - Required skills

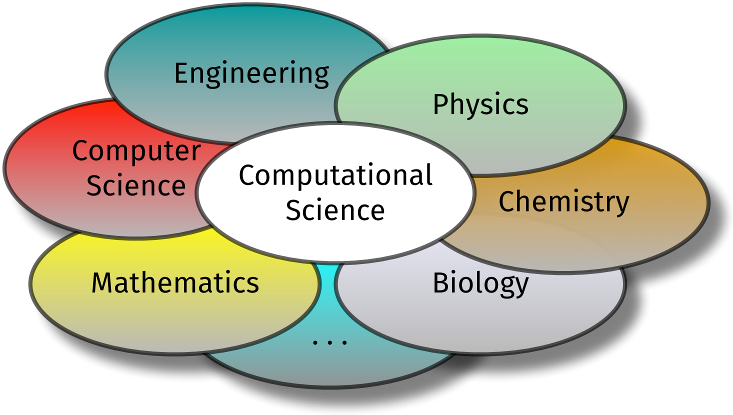
Minimum requirements:

- understanding of application domain
- understanding of programming and data structures

Additional skills to be more efficient:

- overview of existing libraries / tools
- (research) software engineering
- basic understanding of hardware and use through software if performance matters

Computational Science — a new domain



Computational science:
an *enabling methodology*, like literacy and mathematics

Computational Science — a new domain

- Computational Science is not Computer Science
- specific skill set required: application domain knowledge *and* computational science
- often scientists who learn the computational side
- no clear career path: neither scientist nor software engineer
- growing movement to establish such roles in academia: Research Software Engineer
  - `https://www.software.ac.uk`
  - `https://www.de-rse.org`

BETTER
SOFTWARE
BETTER
RESEARCH

- “better software, better research”

This course: Why Python?

- Python is relatively easy to learn [1]
- high efficiency: a few lines of code achieve a lot of computation
- growing use in (open source) academia and industry, thus
- many relevant libraries available
- minimises the time of the programmer
- but: (naive) Python in general much slower execution than compiled languages (such as Fortran, C, C++, Rust, ...).

[1] https://link.springer.com/chapter/10.1007/978-3-540-25944-2_157

This course: Introduction to Python for Computational Science

- introduces the foundations of computational science and data science
- Python programming language
- focus on parts of the Python programming language relevant to computational science
- computational science methodology
- research software engineering
- enable self-directed learning in the future

This course: learning methods

- daily lectures
- daily laboratory sessions (think computer laboratory)
    - opportunity to start and complete self-paced exercises, and to ask for any other clarification.
- automatic feedback on submitted exercises
- teaching materials and lectures are designed to support practical exercises

This course: practicalities

Source of information:

`http://www.desy.de/~fangohr/teaching`

- time table
- laboratory exercises
- pdf files of these slides (may change)
- additional textbook
- further materials

Part 1

Part 1

First steps with Python

Hello World program

- Our first Python program: Entered interactively in Python prompt:

```
>>> print("Hello World")
Hello World
```

Or in Interactive Python (IPython) prompt:

```
In [1]: print("Hello world")
Hello world
```

- Python prompt (`>>>`) and IPython prompt (`In [ ]:`) are very similar
- We prefer the more convenient IPython prompt (but the slides usually show the more compact `>>>` notation)

*Read-Eval-Print Loop (REPL)

The python and the IPython prompt are both examples for a READ-EVAL-PRINT LOOP (REPL):

- Read (the command the user enters)
- Evaluate (the command)
- Print (the result of the evaluation)
- Loop (i.e. go back to the beginning and wait for next command)

Integrated development environments (Spyder)

- You can write programs with a python prompt, a shell and an editor
- More convenient is the use of an “Integrated Development Environment” (IDE)
- Example IDEs: Spyder, Visual Studio Code, PyCharm, IDLE, Emacs, …
- A python prompt is typically embedded in the IDE
- We use Spyder in this module

Everything in Python is an object (with a type)

```
>>> type("Hello World")
<class 'str'>             # "Hello world" is a string
                          # 'class' means 'type'
>>> type(print)
<class 'builtin_function_or_method'>

>>> type(10)
<class 'int'>              # 10 is an integer number

>>> type(3.5)
<class 'float'>            # 3.5 is floating point number
                           # (floating point number:
                           # it has a decimal point)
>>> type('1.0')
<class 'str'>              # string (because of the quotes)

>>> type(1 + 3j)
<class 'complex'>          # complex number
```

Python prompt can act like a calculator

```
>>> 2 + 3
5
>>> 42 - 15.3
26.7
>>> 100 * 11
1100
>>> 2400 / 20
120
>>> 2 ** 3                          # 2 to the power of 3
8
>>> 9 ** 0.5                        # sqrt of 9
3.0
```

Create variables through assignment

```
>>> a = 10
>>> b = 20
>>> a           # short cut for 'print(a)'
10
>>> b           # short cut for 'print(b)'
20
>>> a + b       # ...
30
>>> ab4 = (a + b) / 4
>>> ab4
7.5
```

Functions and using existing functions

- Example: `print` function

```
>>> print("Hello World")
Hello World
```

  The `print` function takes an argument (here a string), and does something with the argument. (Here printing the string to the screen.)

- Example: `abs` function

```
>>> x = -100
>>> y = abs(x)
>>> print(y)
100
```

  A function may return a value: the `abs` function returns the absolute value (100) of the argument (-100).

The `help` function

The `help(x)` function provides documentation for object `x`.

Example:

```
>>> help(abs)
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.
```

Introspection (`dir`)

The directory function (`dir`)

- Everything in Python is an object.
- Python objects have *attributes*.
- `dir(x)` returns the attributes of object `x`
- Example:

```
>>> c = 2 + 1j
>>> dir(c)  # we ignore attributes starting with __
[ ... 'conjugate', 'imag', 'real']
>>> c.imag
1.0
>>> c.real
2.0
>>> c.conjugate()
(2-1j)
```

Attributes of objects can be functions

Example:

```
>>> c = 2 + 1j
>>> dir(c)
[ ... 'conjugate', 'imag', 'real']
>>> type(c.real)
<class 'float'>
>>> type(c.conjugate)
<class 'builtin_function_or_method'>
```

To *execute* a function, we need to add `()` to their name:

```
>>> c.conjugate    # this is the function object
<built-in method conjugate of complex object at 0x10a95f3d0>
>>> c.conjugate()  # this executes the function
(2-1j)             # return value of conjugate function
```

An object attribute that is a function, is called a *method*.

Introspection example with string

```
>>> word = 'test'
>>> print(word)
test
>>> type(word)
<class str>
>>> dir(word)
['__add__', '__class__', '__contains__', ...,
'__doc__', ..., 'capitalize', <snip>,
'endswith', ..., 'upper', 'zfill']
>>> word.upper()
'TEST'
>>> word.capitalize()
'Test'
>>> word.endswith('st')
True
>>> word.endswith('a')
False
```

Summary useful commands (introspection)

- `print(x)` to display the object `x`

  Not needed at the prompt, but in programs that we will write later.
- `type(x)` to determine the type of object `x`
- `help(x)` to obtain the documentation string for object `x`
- `dir(x)` to display the methods and members of object `x`, or the current name space (`dir()`).

Defining functions

Function terminology

Example `abs(x)` function:

```
x = -1.5
y = abs(x)
```

- `x` is the *argument* given to the function (also called *input* or *parameter*)
- `y` is the *return value* (the result of the function's computation)
- Functions may expect zero, one or more arguments
- Not all functions (seem to) return a value. (If no `return` keyword is used, the special object `None` is returned.)

Defining a function ourselves

- Functions
    - provide (potentially complicated) functionality
    - are building blocks of computer programs
    - hide complexity from the user of the function
    - help manage complexity of software

- Example 1:

```
def mysum(a, b):
    return a + b


# main program starts here
print("The sum of 3 and 4 is", mysum(3, 4))
```

Functions should be documented (“docstring”)

```python
def mysum(a, b):
    """Return the sum of parameters a and b."""
    return a + b

# main program starts here
print("The sum of 3 and 4 is", mysum(3, 4))
```

Can now use the help function for our new function:

```
>>> help(mysum)
Help on function mysum in module __main__:

mysum(a, b)
    Return the sum of parameters a and b.
```

Function documentation strings

```python
def mysum(a, b):
    """Return the sum of parameters a and b."""
    return a + b
```

Essential information for documentation string:

- What inputs does the function expect?
- What does the function do?
- What does it return?

*Desirable:

- Examples
- Notes on algorithm (if relevant)
- exceptions that might be raised
- [Author, date, contact details: not needed if version control is used]

Advanced: Recommendations for documentation string style are numpydoc style or PEP257 docstring conventions.

Function documentation string example 1

```python
def mysum(a, b):
    """Return the sum of parameters a and b.

    Parameters
    ----------
    a : numeric
        first input
    b : numeric
        second input

    Returns
    -------
    a+b : numeric
        returns the sum (using the + operator) of a and b. The return type will
        depend on the types of `a` and `b`, and what the plus operator returns.

    Examples
    --------
    >>> mysum(10, 20)
    30
    >>> mysum(1.5, -4)
    -2.5

    Notes
    -----
    History: example first created 2002, last modified 2013
    Hans Fangohr, fangohr@soton.ac.uk,
    """
    return a + b
```

Function documentation string example 2

```python
def factorial(n):
    """Compute the factorial recursively.

    Parameters
    ----------
    n : int
        Natural number `n` > 0 for which the factorial is computed.

    Returns
    -------
    n! : int
        Returns n * (n-1) * (n-2) * ... * 2 * 1

    Examples
    --------
    >>> factorial(1)
    1
    >>> factorial(3)
    6
    >>> factorial(10)
    3628800
    """
    assert n > 0

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

Function example

```
def plus42(n):
"""Add 42 to n and return""" # docstring

    result = n + 42          # body of
    return result            # function

# main program follows
a = 8
b = plus42(a)
```

After execution, `b` carries the value `50` (and a = 8).

Summary functions

- Functions provide (black boxes of) functionality:
  crucial building blocks that hide complexity
- interaction (input, output) through input arguments and return values

  (*printing* and *returning* values is not the same, see slide 41)
- docstring provides the specification (contract) of the function's input, output and behaviour
- advanced*: a function should (normally) not modify input arguments

  (watch out for lists, dicts, more complex data structures as input arguments)

Functions printing vs returning values

Key message: functions should generally *return* values.

We use the Python prompt to explore the difference with these two function definitions:

```python
def print42():
    print(42)

def return42():
    return 42
```

Functions printing vs returning values

```
>>> b = return42()    # return 42, is assigned
>>> print(b)          # to b
42

>>> a = print42()     # return None, and
42                    # print 42 to screen
>>> print(a)
None                  # special object None
```

Functions printing vs returning values

If we use IPython, it shows whether a function returns something (i.e. not None) through the `Out [ ]` token:

```
In [1]: return42()
Out[1]: 42                # Return value of 42



In [2]: print42()
42                        # No 'Out [ ]', so no
                          # returned value
```

Summary: to print or to return?

- A function that returns the control flow through the `return` keyword, will return the object given after `return`.
- A function that does not use the `return` keyword, returns the special object `None`.
- Generally, functions should return a value.
- Generally, functions should not print anything.
- Calling functions from the prompt can cause some confusion here: if the function returns a value and the value is not assigned, it will be printed.

About Python

Python

What is Python?

- High level programming language
- interpreted
- supports three main programming styles (imperative=procedural, object-oriented, functional)
- General purpose tool, yet good for numeric work with extension libraries

Availability

- Python is free
- Python is platform independent (works on Windows, Linux/Unix, Mac OS, ...)
- Python is open source

Which Python version

- There are currently two versions of Python:
    - Python 2.7 and
    - Python 3.x
- Python 2.x and 3.x are incompatible although the changes only affect very few commands.
- Write new programs in Python 3.
- You may have to read / work with Python 2 code at some point.

Python documentation

There is lots of useful documentation:

- Teaching materials on website, including these slides and a text-book like document
- Online documentation, for example
    - http://www.python.org) (Python home page)
    - Matplotlib (publication figures)
    - Numpy (fast vectors and matrices, (NUMerical PYthon)
    - SciPy (scientific algorithms, `solve_ivp`)
    - Pandas (data engineering and data science)
    - scikit-learn (machine learning)
    - SymPy (Symbolic calculation)
- interactive documentation (such as `dir()` and `help()`)

Using modules

The math module (`import math`)

```
>>> import math
>>> math.sqrt(4)
2.0
>>> math.pi
3.141592653589793
>>> dir(math)       #attributes of 'math' object
['__doc__', '__file__', < snip >
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil', 'copysign', 'cos', 'e', 'erf',
'exp', <snip>, 'sqrt', 'tan', 'tanh', 'trunc']

>>> help(math.sqrt)       # ask for help on sqrt
sqrt(...)
    sqrt(x)
    Return the square root of x.
```

Name spaces and modules

Three (good) options to access a module:

1. use the full name:

```python
import math
print(math.sin(0.5))
```

2. use some abbreviation

```python
import math as m
print(m.sin(0.5))
print(m.pi)
```

3. import all objects we need explicitly

```python
from math import sin, pi
print(sin(0.5))
print(pi)
```

Modules provide functionality

- each module provides some additional python functionality
- Python has many modules:
  - *Python Standard Library*: `math`, `pathlib`, `sys`, ...
  - Contributions from others: `numpy`, `jupyter`, `pytest`, ...
  - Every programmer can create their own modules.
- there is distinction between *module*, *package*, and *library* but in practice the terms are used interchangeably.

Conditionals, if-else

Truth values

The python values `True` and `False` are special inbuilt objects:

```
>>> a = True
>>> print(a)
True
>>> type(a)
<class bool>
>>> b = False
>>> print(b)
False
>>> type(b)
<class bool>
```

Truth values

We can operate with these two logical values using boolean logic, for example the logical and operation (`and`):

```
>>> True and True          # logical and operation
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

Truth values

There is also logical or (`or`) and the negation (`not`):

```
>>> True or False
True
>>> not True
False
>>> not False
True
>>> True and not False
True
```

Truth values

In computer code, we often need to evaluate some expression that is either true or false (sometimes called a “predicate”). For example:

Truth values

```
>>> x = 30        # assign 30 to x
>>> x >= 30       # is x greater than or equal to 30?
True
>>> x > 15        # is x greater than 15
True
>>> x > 30
False
>>> x == 30       # is x the same as 30?
True
>>> not x == 42   # is x not the same as 42?
True
>>> x != 42       # is x not the same as 42?
True
```

if-then-else

The `if-else` command allows to branch the execution path depending on a condition. For example:

```
>>> x = 30                  # assign 30 to x
>>> if x > 30:              # predicate: is x > 30
...     print("Yes")            # if True, do this
... else:
...     print("No")             # if False, do this
...
No
```

if-then-else

The general structure of the `if-else` statement is

```
if A:
    B
else:
    C
```

where `A` is the predicate.

- If `A` evaluates to `True`, then all commands `B` are carried out (and `C` is skipped).
- If `A` evaluates to `False`, then all commands `C` are carried out (and `B`) is skipped.
- `if` and `else` are Python keywords.

if-then-else

`A` and `B` can each consist of multiple lines, and are grouped through indentation as usual in Python.

if-else example

```python
def slength1(s):
    """Returns a string describing the
    length of the sequence s"""
    if len(s) > 10:
        ans = 'very long'
    else:
        ans = 'normal'

    return ans
```

```
>>> slength1("Hello")
'normal'
>>> slength1("HelloHello")
'normal'
>>> slength1("Hello again")
'very long'
```

if-elif-else example

If more cases need to be distinguished, we can use the keyword `elif` (standing for ELse IF) as many times as desired:

```python
def slength2(s):
    if len(s) == 0:
        ans = 'empty'
    elif len(s) > 10:
        ans = 'very long'
    elif len(s) > 7:
        ans = 'normal'
    else:
        ans = 'short'

    return ans
```

if-elif-else example

```
>>> slength2("")
'empty'
>>> slength2("Good Morning")
'very long'
>>> slength2("Greetings")
'normal'
>>> slength2("Hi")
'short'
```

LAB1

Raising exceptions

Exceptions

- Errors arising during the execution of a program result in “exceptions” being ’raised’ (or ’thrown’).
- We have seen exceptions before, for example when dividing by zero:

```
>>> 4.5 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division by zero
```

or when we try to access an undefined variable:

Exceptions

```
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

- Exceptions are a modern way of dealing with error situations
- We will now see
    - what exceptions are coming with Python
    - how we can raise ("throw") exceptions in our code

In-built Python exceptions

Python's in-built exceptions (from https://docs.python.org/3/library/exceptions.html)

In-built Python exceptions

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
```

Raising exceptions

- Because exceptions are Python's way of dealing with runtime errors, we should use exceptions to report errors that occur in our own code.
- To raise a `ValueError` exception, we use

  ```
  raise ValueError("Message")
  ```

  and can attach a message `"Message"` (of type string) to that exception which can be seen when the exception is reported or caught:

  ```
  >>> raise ValueError("Some problem occurred")
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  ValueError: Some problem occurred
  ```

Raising NotImplementedError Example

Often used is the `NotImplementedError` in *incremental software development*:

```python
def my_complicated_function(x):
    message = f"Called with x={x}"
    raise NotImplementedError(message)
```

If we call the function:

```
>>> my_complicated_function(42)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in my_complicated_function
NotImplementedError: Called with x=42
```

Sequences

Sequences overview

Different types of sequences

- strings
- lists (mutable)
- tuples (immutable)
- arrays (mutable, part of numpy)

They share common behaviour.

Strings

```
>>> a = "Hello World"
>>> type(a)
<class str>
>>> len(a)
11
>>> print(a)
Hello World
```

Different possibilities to limit strings:

```
  'A string'
  "Another string"
  "A string with a ' in the middle"
  """A string with triple quotes can
  extend over several
  lines"""
```

Strings 2 (exercise)

- Define a, b and c at the Python prompt:

```
>>> a = "One"
>>> b = "Two"
>>> c = "Three"
```

- Exercise: What do the following expressions evaluate to?
    1. `d = a + b + c`
    2. `5 * d`
    3. `d[0], d[1], d[2]` (indexing)
    4. `d[-1]`
    5. `d[4:]` (slicing)

Strings 3 (exercise)

```
>>> s="""My first look at Python was an
... accident, and I didn't much like what
... I saw at the time."""
```

For the string s:

- count the number of (i) letters 'e' and (ii) substrings 'an'
- replace all letters 'a' with '0'
- make all letters uppercase
- make all capital letters lowercase, and all lower case letters to capitals

Lists

```
[]                         # the empty list
[42]                       # a 1-element list
[5, 'hello', 17.3]         # a 3-element list
[[1, 2], [3, 4], [5, 6]] # a list of lists
```

- Lists store an ordered sequence of Python objects
- Access through index (and slicing) as for strings.
- use **`help()`**, often used list methods is **`append()`**

(In general computer science terminology, vector or array might be better name as the actual implementation is not a linked list, but direct $\mathcal{O}(1)$ access through the index is possible.)

Example program: using lists

```
>>> a = []                  # creates a list
>>> a.append('dog')         # appends string 'dog'
>>> a.append('cat')         # ...
>>> a.append('mouse')
>>> print(a)
['dog', 'cat', 'mouse']
>>> print(a[0])             # access first element
dog                         # (with index 0)
>>> print(a[1])             # ...
cat
>>> print(a[2])
mouse
>>> print(a[-1])            # access last element
mouse
>>> print(a[-2])            # second last
cat
```

Example program: lists containing a list

```
>>> a = ['dog', 'cat', 'mouse', [1, 10, 100, 1000]]
>>> a
['dog', 'cat', 'mouse', [1, 10, 100, 1000]]
>>> a[0]
dog
>>> a[3]
[1, 10, 100, 1000]
>>> max(a[3])
1000
>>> min(a[3])
1
>>> a[3][0]
1
>>> a[3][1]
10
>>> a[3][3]
1000
```

Sequences – more examples

```
>>> a = "hello world"
>>> a[4]
'o'
>>> a[4:7]
'o w'
>>> len(a)
11
>>> 'd' in a
True
>>> 'x' in a
False
>>> a + a
'hello worldhello world'
>>> 3 * a
'hello worldhello worldhello world'
```

Tuples

- tuples are very similar to lists
- tuples are *immutable* (unchangeable after they have been created) whereas lists are *mutable* (changeable)
- tuples are usually written using parentheses ($\leftrightarrow$ “round brackets”):

Tuples

```
>>> t = (3, 4, 50)    # t for Tuple
>>> t
(3, 4, 50)
>>> type(t)
<class tuple>

>>> L = [3, 4, 50]    # compare with L for List
>>> L
[3, 4, 50]
>>> type(L)
<class list>
```

Tuples are defined by comma

- tuples are defined by the comma (!), not the parenthesis

```
>>> a = 10, 20, 30
>>> type(a)
<class tuple>
```

- the parentheses are usually optional (but should be written anyway):

```
>>> a = (10, 20, 30)
>>> type(a)
<class tuple>
```

Tuples are sequences

- normal indexing and slicing (because tuple is a sequence)

```
>>> t[1]
4
>>> t[:-1]
(3, 4)
```

Why do we need tuples (in addition to lists)?

1. use tuples if you want to make sure that a set of objects doesn't change.
2. Using tuples, we can assign several variables in one line (known as *tuple packing* and *unpacking*)

```
x, y, z = 0, 0, 1
```

This allows "instantaneous swap" of values:

```
a, b = b, a
```

Strictly: "tuple packing" on right hand side and "sequence unpacking" on left.

Why do we need tuples (in addition to lists)?

3. functions return tuples if they return more than one object

```
def f(x):
    return x**2, x**3

a, b = f(x)
```

4. tuples can be used as keys for dictionaries as they are immutable

(Im)mutables

- Strings — like tuples — are immutable:

```
>>> a = 'hello world'          # String example
>>> a[3] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object does not support item assignment
```

- strings can only be ’changed’ by creating a new string, for example:

```
>>> a = a[0:3] + 'x' + a[4:]
>>> a
'helxo world'
```

Summary sequences

- lists, strings and tuples (and arrays) are sequences.
- sequences share the following operations

`a[i]`	returns element with index *i* of `a`
`a[i:j]`	returns elements *i* up to $j-1$
`len(a)`	returns number of elements in sequence
`min(a)`	returns smallest value in sequence
`max(a)`	returns largest value in sequence
`x in a`	returns `True` if `x` is element in `a`
`a + b`	concatenates `a` and `b`
`n * a`	creates `n` copies of sequence `a`

In the table above, `a` and `b` are sequences, `i`, `j` and `n` are integers, `x` is an element.

Conversions of sequence to list and to tuple

- to tuple:
  Convert any sequence into a tuple using the `tuple` function:

```
>>> tuple([1, 4, "dog"])
(1, 4, 'dog')
```

- to list:
  Convert any sequence into a list using the `list` function:

```
>>> list((10, 20, 30))
[10, 20, 30]
```

Conversions of sequence to strings

- every string object `s` has a `join` method that *joins* elements of a squence together, with the string `s` connecting the sequence elements:

```
>>> x = ['A', 'list', 'of', 'strings.']
>>> " ".join(x)
'A list of strings.'
>>> "-".join(x)
'A-list-of-strings.'
>>> "-um-".join(x)
'A-um-list-um-of-um-strings.'
>>> "".join(x)
'Alistofstrings.'
```

- Only works if the elements in the sequence are of type string already:

```
>>> a = [10, 20, 30]
>>> "-".join(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 0: expected str instance, int found
>>>
```

*Conversions to and from iterators

- *Looking ahead* to iterators, we note that `list` and `tuple` can also convert from iterators:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

- *And if you ever need to create an iterator from a sequence, the `iter` function can this:

```
>>> iter([1, 2, 3])
<list_iterator object at 0x1013f1fd0>
```

Reversing a sequence with slicing operator `::-1`

- The slicing operator `::-1` creates a *reversed copy* of a sequence:

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]     # list
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

>>> "Hello World"[::-1]                     # string
'dlroW olleH'
```

- Why does this work?

```
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[:]   # slice from beginning to end (creates copy of a)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a[::2]  # slice from beginning to end in steps of 2
[0, 2, 4, 6, 8]
>>> a[::-2]  # in steps of -2
[9, 7, 5, 3, 1]
>>> a[::-1]  # in steps of -1
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Reversing a list with `list.reverse()` method

- `list` objects have an in-built `reverse()` method:

```
>>> a = [1, 2, 3, 4]
>>> a.reverse()
>>> a
[4, 3, 2, 1]
>>> tuple([1, 4, "dog"])
```

- this is called working "in place" as it re-arranges the data in the place where it is stored (in contrast to creating a second copy)
- useful if the data is large and we want to avoid a second copy
- not available for string and tuple as these are immutable

*Reverse sequence with `reversed` iterator

```
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> b = reversed(a)
>>> b   # will iterate through
        # reverse sequence when needed
<list_reverseiterator object at 0x101117d30>
>>> type(b)
<class 'list_reverseiterator'>
>>> list(b)   # conversion to list enforces iteration
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> list(reversed(a))   # reversing a in one line
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Loops

Introduction loops

Computers are good at repeating tasks (often the same task for many different sets of data).

Loops are the way to execute the same (or very similar) tasks repeatedly (“in a loop”).

Python provides the “for loop” and the “while loop”.

Example program: for-loop

```
animals = ['dog', 'cat', 'mouse']

for animal in animals:
    print(f"This is the {animal}!")
```

produces

```
This is the dog!
This is the cat!
This is the mouse!
```

The for-loop *iterates* through the sequence `animals` and assigns the values in the sequence subsequently to the name `animal`.

Iterating over integers

Often we need to iterate over a sequence of integers:

```python
for i in [0, 1, 2, 3, 4, 5]:
    print(f"the square of {i} is {i**2}")
```

produces

```
the square of 0 is 0
the square of 1 is 1
the square of 2 is 4
the square of 3 is 9
the square of 4 is 16
the square of 5 is 25
```

Iterating over integers with `range`

The `range(n)` object is used to iterate over a sequence of increasing integer values up to (but not including) `n`:

```python
for i in range(6):
    print(f"the square of {i} is {i**2}")
```

produces

```
the square of 0 is 0
the square of 1 is 1
the square of 2 is 4
the square of 3 is 9
the square of 4 is 16
the square of 5 is 25
```

The range object

- `range` is used to iterate over integer sequences
- We can use the range object in for loops:

```
>>> for i in range(3):
...     print(f"i={i}")
i=0
i=1
i=2
```

The range object

- We can convert it to a list:

```
>>> list(range(6))
[0, 1, 2, 3, 4, 5]
```

- This conversion to list is useful to understand what sequences the range object would provide if used in a for loop:

The range object

```
>>> list(range(6))
[0, 1, 2, 3, 4, 5]
>>> list(range(0, 6))
[0, 1, 2, 3, 4, 5]
>>> list(range(3, 6))
[3, 4, 5]
>>> list(range(-3, 0))
[-3, -2, -1]
```

The range object

- *Advanced: `range` has its own type:

```
>>> type(range(6))
<class range>
```

`range` objects are lazy sequences (Python range is not an iterator)

Summary `range`

`range`

`range([start,] stop [,step])` iterates over integers from `start` up to to `stop` (*but not including* `stop`) in steps of `step`.

`start` defaults to 0 and `step` defaults to 1.

```
>>> list(range(0, 10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
>>> list(range(5, 4))
[]                        # no iterations
```

Iterating over sequences with `for`-loop

- `for` loop iterates over iterables.
- Sequences are iterable.
- Examples

```python
for i in [0, 3, 4, 19]:            # list is a
    print(i)                       # sequence

for animal in ['dog', 'cat', 'mouse']:
    print(animal)

for letter in "Hello World":       # strings are
    print(letter)                  # sequences

for i in range(5):                 # range objects
    print(i)                       # are iterable
```

Example: create list with for-loop

```python
def create_list_of_increasing_halfs(n):
    """Given integer n >=0, return list of length
    n starting with [0, 0.5, 1.0, 1.5, ...]."""
    result = []
    for i in range(n):
        number = i * 1 / 2
        result.append(number)
    return result

# main program
print(create_list_of_increasing_halfs(5))
```

Output:

```
[0.0, 0.5, 1.0, 1.5, 2.0]
```

Example: modify list with for-loop

```
def modify_list_add_42(original_list):
    """Given a list, add 42 to every element
    and return"""
    modified_list = []
    for element in original_list:
        new_element = element + 42
        modified_list.append(new_element)
    return modified_list

# main program
print(modify_list_add_42([0, 10, 100, 1000]))
```

Output:

```
[42, 52, 142, 1042]
```

Reminder: If-then-else

- Example 1 (if-then-else)

```
a = 42
if a > 0:
    print("a is positive")
else:
    print("a is negative or zero")
```

Another iteration example

This example generates a list of numbers often used in hotels to label floors (more info)

```python
def skip13(a, b):
    """Given ints a and b, return
    list of ints from a to b without 13"""
    result = []
    for k in range(a, b):
        if k == 13:
            pass  # do nothing
        else:
            result.append(k)
    return result
```

Another iteration example (with `continue`)

This example generates a list of numbers often used in hotels to label floors (more info)

```
def skip13(a, b):
    """Given ints a and b, return
    list of ints from a to b without 13"""
    result = []
    for k in range(a, b):
        if k == 13:
            continue  # jump to next iteration
        result.append(k)
    return result
```

Exercise range_double

Write a function `range_double(n)` that generates a list of numbers similar to `list(range(n))`. In contrast to `list(range(n))`, each value in the list should be multiplied by 2. For example:

```
>>> range_double(4)
[0, 2, 4, 6]
>>> range_double(10)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

For comparison the behaviour of `range`:

```
>>> list(range(4))
[0, 1, 2, 3]
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

For loop summary

- `for`-loop to iterate over sequences
- can use `range` to generate sequences of integers
- special keywords:
    - `continue` - skip remainder of body of statements and continue with next iteration
    - `break` - leave for-loop immediately
- *Advanced:
    - can iterate over any *iterable*
    - we can create our own iterables
    - See summary Socratica on Iterators, Iterables, and Itertools

Exercise: First In First Out (FIFO) queue

Write a *First-In-First-Out* queue implementation, with functions:

- `add(name)` to add a customer with name `name` (call this when a new customer arrives)
- `next()` to be called when the next customer will be served. This function returns the name of the customer
- `show()` to print all names of customers that are currently waiting
- `length()` to return the number of currently waiting customers

Suggest to use a global variable `q` and define this in the first line of the file by assigning an empty list: `q = []`.

While loops

- Reminder:
  a `for` loop iterates *over a given sequence* or iterable
- A `while` loop iterates *while a condition is fulfilled*

-
  ```
  x = 64
  while x > 10:
      x = x // 2
      print(x)
  ```

  produces

  ```
  32
  16
  8
  ```

*While loop example 2

Determine ϵ:

```
eps = 1.0


while eps + 1 > 1:
    eps = eps / 2.0
print(f"epsilon is {eps}")
```

Output:

```
epsilon is 1.11022302463e-16
```

Style guide for Python code

Syntax versus style

- Python programs *must* follow Python syntax.
- Python programs *should* follow Python style guide, because
    - readability is key (debugging, documentation, team effort)
    - conventions improve effectiveness

Common style guide: PEP8

From http://www.python.org/dev/peps/pep-0008/:

- This style guide evolves over time as additional conventions are identified and past conventions are rendered obsolete by changes in the language itself.
- *”Readability counts”*: One of Guido van Rossum’s key insights is that code is *read much more often than it is written*. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code.

PEP8 Style guide

- Indentation: use 4 spaces
- One space around assignment operator (=) operator: `c = 5` and not `c=5`.
- Spaces around arithmetic operators can vary. Both `x = 3*a + 4*b` and `x = 3 * a + 4 * b` are okay.
- No space before and after parentheses: `x = sin(x)` but not `x = sin( x )`
- A space after comma: `range(5, 10)` and not `range(5,10)`.
- No whitespace at end of line
- No whitespace in empty line
- One or no empty line between statements within function

PEP8 Style guide

- Two empty lines between functions
- One import statement per line
- import first standard Python library (such as `math`), then third-party packages (`numpy`, `scipy`, ...), then our own modules
- no spaces around = when used in keyword arguments: `"Hello World".split(sep=' ')` but not `"Hello World".split(sep = ' ')`

PEP8 Style Summary

- Follow PEP8 guide, in particular for new code.
- Use tools to help us:
    - Spyder editor can show PEP8 violations (In Spyder 6: `Preferences` → `Completion and Linting` → `Code style and formatting` → `[X] Enable code style lintiing` → `[OK]`)
    - Similar tools/plugins are available for other editors. editors.
    - `pycodestyle` program available to check source code from command line (used to be called `pep8` in the past).
      To check file `myfile.py` for PEP8 compliance:

      ```
      pycodestyle myfile.py
      ```

*Style conventions for *documentation strings*

- Python documentation strings (pydoc) conventions:
  - PEP257 docstring style (from 2001), basis for both
  - numpydoc style (science) and
  - Google pydoc style
- Examples on slide 33 and 34 are compatible with all conventions
- Editors can highlight deviations
- Program to check documentation string style compliance in file `myfile.py`:

  -
    ```
    pydocstyle --convention=pep257 myfile.py
    ```

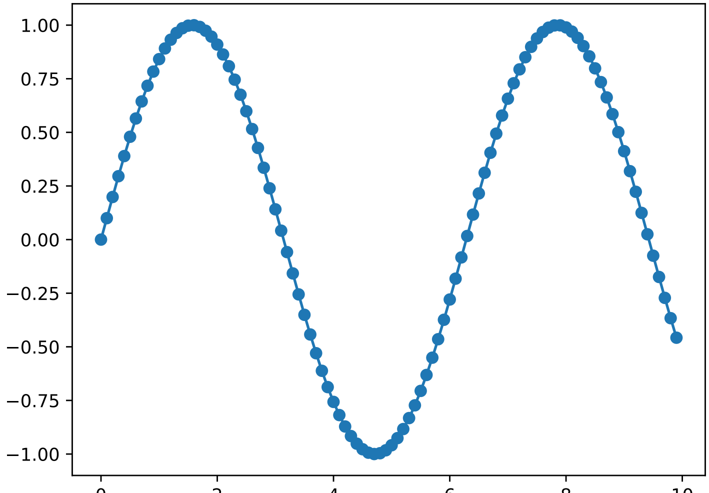
  -
    ```
    pydocstyle --convention=numpy myfile.py
    ```

  -
    ```
    pydocstyle --convention=google myfile.py
    ```

Outlook: first plot

```python
import math
import matplotlib.pyplot as plt  # convention

xs = []  # store x-values for plot in list
ys = []  # store y-values for plot in list
for i in range(100):  # compute data
    x = 0.1 * i
    xs.append(x)
    y = math.sin(x)  # we plot sin(x)
    ys.append(y)

# plot data
plt.plot(xs, ys, '-o')

plt.savefig("matplotlib-mini-example.pdf")
```

Outlook: first plot



Reading and writing files

File input/output

It is a common task to

- read some input data file
- do some calculation/filtering/processing with the data
- write some output data file with results

Python distinguishes between

- *text* files (`'t'`)
- *binary* files `'b'`)

If we don’t specify the file type, Python assumes we mean text files.

Writing a text file

```
>>> with open('test.txt', 'tw') as f:
...     f.write("first line\nsecond line")
...
22
```

creates a file `test.txt` that reads

```
first line
second line
```

Writing a text file

- To write data, we need to open the file with `'w'` mode:

  ```
      with open('test.txt', 'w') as f:
  ```

  By default, Python assumes we mean text files. However, we can be explicit and say that we want to create a Text file for Writing:

  ```
      with open('test.txt', 'wt') as f:
  ```

- If the file exists, it will be overridden with an empty file when the open command is executed.
- The file object `f` has a method `f.write` which takes a string as an input argument.

Reading a text file

We create a file object `f` using

```
>>> with open('test.txt', 'rt') as f:   # Read Text
```

and have different ways of reading the data:

1. `f.read()` returns one long string for the whole file

```
>>> with open('test.txt', 'rt') as f:
...     data = f.read()
...
>>> data
'first line\nsecond line'
```

Reading a text file

2. `f.readlines()` returns a list of strings (each being one line)

```
>>> with open('test.txt', 'rt') as f:
...     lines = f.readlines()
...
>>> lines
['first line\n', 'second line']
```

Reading a text file

3. *Advanced: Use text file `f` as an iterable object: process one line in each iteration

```
>>> with open('test.txt', 'rt') as f:
>>>     for line in f:
...         print(line, end='')
...
first line
second line
>>> f.close()
```

Reading a text file

This is important for large files: the file can be larger than the computer RAM as only one line at a time is read from disk to memory.

*File input and output without context manager

With file context manager (recommended):

```
>>> with open('test.txt', 'rt') as f:   # This creates
...                                     # the context.
...     data = f.read()                 # We can use 'f'
...                                     # in the context.
...                        # File 'f' is automatically closed
>>> data                   # when the context is left.
'first line\nsecond line'
```

Without file context manager (not recommended!):

```
>>> f = open('test.txt', 'rt')
>>> data = f.read()
>>> f.close()  # must close file manually
>>> data
'first line\nsecond line'
```

Use case: Reading a file, iterating over lines

Often we want to process line by line. Typical code fragment:

```python
with open('myfile.txt', 'rt') as f:
    lines = f.readlines()

# some processing of the lines object
for line in lines:
    print(line)
```

Splitting a string

- We often need to split a string into smaller parts: use string method `split()`:
  (try `help("".split)` at the Python prompt for more info)

Example:

```
>>> c = 'This is my string'
>>> c.split()
['This', 'is', 'my', 'string']
>>> c.split('i')
['Th', 's ', 's my str', 'ng']
```

Useful functions processing text files:

- `string.strip()` method gets rid of leading and trailing white space, i.e. spaces, newlines (`\n`) and tabs (`\t`):

```
>>> a = "   hello\n "
>>> a.strip()
'hello'
```

- `int()` and `float` convert strings into numbers (if possible)

```
>>> int("42")
42
>>> float("3.14")
3.14
>>> int("0.5")
Traceback (most recent call last):
  ValueError: invalid literal for int()
                 with base 10: '0.5'
```

Exercise: Shopping list

Given a list

```
bread       1  1.39
tomatoes    6  0.26
milk        3  1.45
coffee      3  2.99
```

Write program that computes total cost per item, and writes to `shopping_cost.txt`:

```
bread       1.39
tomatoes    1.56
milk        4.35
coffee      8.97
```

One solution

One solution is `shopping_cost.py`

```python
with open('shopping.txt', 'tr') as fin:        # INput File
    lines = fin.readlines()

with open('shopping_cost.txt', 'tw') as fout: # OUTput File
    for line in lines:
        words = line.split()
        itemname = words[0]
        number = int(words[1])
        cost = float(words[2])
        totalcost = number * cost
        fout.write(f"{itemname:10} {totalcost}\n")
```

Exercise

Write function `print_line_sum_of_file(filename)` that reads a file of name `filename` containing numbers separated by spaces, and which computes and prints the sum for each line. A data file might look like

```
    2 3 5 -30 100
    0 45 3 2
    17
```

*Binary files 1

- Files that store *binary* data are opened using the `'b'` flag (instead of `'t'` for Text):

```
        open('data.dat', 'br')
```

- For text files, we read and write `str` objects. For binary files, use the `bytes` type instead.
- By default, store data in text files. Text files are human readable (that's good) but take more disk space than binary files.
- Reading and writing binary data is outside the scope of this introductory module. If you read arbitrary binary data, you may need the `struct` module.
- For large/complex scientific data, consider HDF5.

*HDF5 files

- If you need to store large and/or complex data, consider the use of HDF5 files: `https://portal.hdfgroup.org/display/HDF5/HDF5`
- Python interface: `https://www.h5py.org` (`import h5py`)
- hdf5 files
  - provide a hierarchical structure (like subdirectories and files)
  - can compress data on the fly
  - supported by many tools
  - standard in some areas of science
  - optimised for large volume of data and effective access

Writing modules

Writing module files

- Motivation: it is useful to bundle functions that are used repeatedly and belong to the same subject area into one module file (also called “library”)
- This allows to re-use the functions in multiple Python applications.
- Every Python file can be imported as a module.
- If the module file contains commands (other than class and function *definitions*) then these are executed when the file is imported. This can be desired but sometimes it is not.

The internal `__name__` variable (1)

- Here is an example of a module file saved as `module1.py`:

```
def someusefulfunction():
    pass


print(f"My name is {__name__}")
```

  We can execute this module file, and the output is

```
My name is __main__
```

- The internal variable `__name__` takes the (string) value `"__main__"` if the program file `module1.py` is executed.

The internal `__name__` variable (1)

- On the other hand, we can *import* `module1.py` in another file, for example like this:

  ```
  import module1
  ```

  The output is now:

  ```
  My name is module1
  ```

- We see that `__name__` inside a module takes the value of the module name if the file is imported.

if `__name__ == __main__` …

`module2.py`:

```
1  def someusefulfunction():
2      pass
3
4  if __name__ == "__main__":
5      print("I am the top level")
6  else:
7      print(f"I am imported as a library '{__name__}'")
```

- Line 5 is only executed when the module is executed as the top level (for example as `python module2.py`, or pressing F5 in Spyder when editing the dile `module2.py`).
- `__name__` allows conditional execution of code when top-level or imported.

Application file example

```python
def useful_function():
    # Core function in this app.
    # Could be useful in other apps.
    pass

def main():
    # Main functionality of this app in here.
    useful_function()
    # ...

if __name__ == "__main__":
    main()  # start main application
else:
    # get here if the file is imported
    pass
```

Library file example

```python
def useful_function():
    # core functionality of library here
    pass

def test_for_useful_function():
    print("Running self test ...")

if __name__ == "__main__":
    test_for_useful_function()
else:
    print("Setting up library")
    # initialisation code that might be needed
    # if imported as a library
```

Name spaces, global and local variables

Name spaces — what can be seen where?

We distinguish between

- *global* variables (defined in main program) and
- *local* variables (defined for example in functions)
- *built-in* commands

Python’s look up rule

Python’s look up rule for Names

When coming across an identifier, Python looks for this in the following order in

- the local name space (L)
- (if appropriate in the next higher level local name space), (L^2, L^3, …)
- the global name space (G)
- the set of built-in commands (B)

This is summarised as “LGB” or “L^nGB”.

If the identifier cannot be found, a `NameError` is raised.

Local names shadow global names

- This means, we can *read* global variables from functions. Example:

```python
def f():
    print(x)

x = 'I am global'
f()
```

Output:

```
I am global
```

Local names shadow global names

- but local variables “shadow” global variables:

```
def f():
    y = 'I am local y'
    print(x)
    print(y)

x = 'I am global x'
y = 'I am global y'
f()
print("back in main:")
print(y)
```

Output:

Local names shadow global names

```
I am global x
I am local y
back in main:
I am global y
```

Why should I care about global variables?

- Generally, the use of global variables is not recommended:
    - functions should take all necessary input as arguments
    - and return all relevant output.
    - This makes the functions work as independent units and is essential to control complexity of software (good engineering practice)
- However, sometimes the same constant or variable (such as the mass of an object) is required throughout a program:
    - it is not good practice to define this variable more than once (it is likely that we assign different values and get inconsistent results)

Why should I care about global variables?

- in this case — in small programs — the use of (read-only) global variables may be acceptable.
- Object Oriented Programming provides a somewhat neater solution to this.

Plotting data from csv file

Data analysis example: temperature anomaly

- National Oceanic and Atmospheric Administration (NOAA) hosts climate data at `https://www.ncei.noaa.gov/access/monitoring/climate-at-a-glance/global/time-series/globe/tavg/land_ocean/12/9/1850-2024`
- provides average global temperature data since 1850
- we choose 12-month average from September to August from 1850 to 2024 -> Download CSV
- *anomaly* data shows the *temperature deviation* from the average 1910 to 2000.

Beginning of data file

```
# Title: Land and Ocean Oct - Sept Average Temp Anomalies
# Units: Degrees Celsius
# Base Period: 1901-2000
# Missing: -999
Year,Anomaly
1851,-0.14
1852,-0.07
1853,-0.07
1854,-0.11
1855,-0.06
1856,-0.11
1857,-0.23
1858,-0.17
1859,-0.09
1860,-0.15
1861,-0.32
```

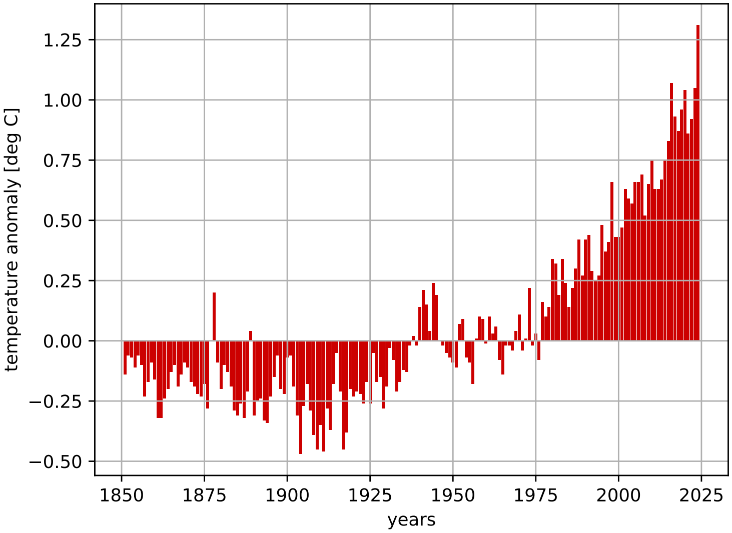
Data analysis example: attempt 1

```
import matplotlib.pyplot as plt

# read data
with open("data.csv", "tr") as f:
    lines = f.readlines()

year = []
dT = []

for line in lines[5:]:  # skip first 5 lines
    a, b = line.split(",")
    year.append(int(a))  # convert string of year to int
    dT.append(float(b))  # convert string of temp to float
```

Data analysis example: attempt 1

```
# plot data
plt.bar(year, dT, color=[0.8, 0, 0])
plt.ylabel("temperature anomaly [deg C]")
plt.xlabel("years")
plt.grid(True)
plt.savefig("anomaly1.pdf")
```

Data analysis example: attempt 1



*Data analysis example: outlook

- In this example, we store each data set in a `list`. Better options are `numpy.array` or `pandas.Series`.
- Here we read the CSV file *manually* but there are dedicated libraries to read CSV files.

  Preferred option is `read_csv()` from `pandas` (259).

```python
import pandas
d = pandas.read_csv('data.txt', skiprows=4, index_col=0)
d.plot.bar()
```

  Next best is `loadtxt()` from `numpy` (221).

```python
import numpy as np
import matplotlib.pyplot as plt
data = np.loadtxt("data.csv", delimiter=",", skiprows=5)
plt.plot(data[:,0], data[:,1])  # axes are not annotated yet
```

Catching exceptions

Exceptions example

- suppose we try to read data from a file:

```
with open('myfilename.txt', 'r') as f:
    lines = f.readlines()
for line in lines:
    print(line)
```

- If the file doesn't exist, then the `open()` function raises the `FileNotFoundError` exception:

```
FileNotFoundError: [Errno 2] No such file or
↪  directory: 'myfilename.txt'
```

Catching exceptions

- We can modify our code to ’catch’ this error:

```python
try:
    with open('myfilename.txt', 'r') as f:
        lines = f.readlines()
except FileNotFoundError:
    print("The file couldn't be found.")
else:
    # this is executed if no exception is raised
    for line in lines:
        print(line)

```

which produces this message:

Catching exceptions

```
The file couldn't be found.
```

- The `try` branch (line 1) will be executed.
- Should an `FileNotFoundError` exception be raised, then the except branch (starting line 4) will be executed.
- Should no exception be raised in the `try` branch, then the `except` branch is ignored, and the program carries on starting in line .

Catching exceptions

Slight extension to print more detailed error message:

```python
try:
    with open('myfilename.txt', 'r') as f:
        lines = f.readlines()
except FileNotFoundError as error:
    print("The file couldn't be found.")
    print(f"Error message: {error}")
else:
    # this is executed if no exception is raised
    for line in lines:
        print(line)

```

Catching exceptions

Output:

```
The file couldn't be found.
Error message: [Errno 2] No such file or directory:
↪  'myfilename.txt'
```

Catching exceptions summary

- Catching exceptions allows us to take action on errors that occur
    - For the file-reading example, we could ask the user to provide another file name if the file can’t be opened.
- Catching an exception once an error has occurred may be easier than checking beforehand whether a problem will occur (“*It is easier to ask forgiveness than get permission*”.)

Overview try-except-else-finally

```python
try:
    # statement that might raise an exception
    pass
except SomeError:
    # deal with error
    pass
else:
    # code to execute if no error is raised
    pass
finally:
    # code that must always be executed
    # (for example closing a file)
    pass
```

try-except example

From Python documentation

```
try:
    f = open("myfile.txt")
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")
except Exception as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

The last `raise` re-raises the last exception as if it wasn't caught before.

Exercise

Extend `print_line_sum_of_file(filename)` so that if the data file contains non-numbers (i.e. strings), these evaluate to the value 0. For example

```
    1 2 4        -> 7
    1 cat 4      -> 5
    coffee       -> 0
```

Print

`print` function

- the `print` function sends content to the "standard output" (usually the screen)
- `print()` prints an empty line:

```
>>> print()

```

- Given a single string argument, this is printed, followed by a new line character:

```
>>> print("Hello")
Hello
```

print function

- Given multiple objects separated by commas, they will be printed separated by a space character:

```
>>> print("dog", "cat", 42)
dog cat 42
```

- To supress printing of a new line, use the end option:

```
>>> print("Dog", end=""); print("Cat")
DogCat
>>>
```

`print` function

- Given another object (not a string), the `print` function will *ask* the object for its preferred way to be represented as a string (via the `__str__` method):

```
>>> print(42)
42
```

  With Object Oriented programming, we can customise the `__str__` method for each class.

Common strategy for the print command

- Construct some string `s`, then print this string using the `print` function

```
>>> s = "I am the string to be printed"
>>> print(s)
I am the string to be printed
```

- The question is, how can we construct the string `s`? We talk about string formatting.

String formatting

String formatting & Example 1

- Task: Given some objects, we would like to create a string representation.
- Example 1: a variable `t` has the value 42.123 and we like to print `Duration is 42.123s` to the screen.
- Solution: Create a *formatted string* "`Duration is 42.123s`" and pass this string to the `print` function:

```
>>> t = 42.123
>>> print(f"Duration = {t}s")
Duration = 42.123s
```

- With *string formatting*, we mean the creation of the string "`Duration is 42.123s`"

String formatting & Example 2

- Example 2: a variable `t` has the value 42.123 and we like to print `Duration is 42.1s` to the screen (i.e round to one post-decimal digit.)
- Solution:

```
>>> t = 42.123
>>> print(f"Duration = {t:.1f}s")
Duration = 42.1s
```

String formatting: Example 2 explanation

Explanation of `f"Duration = {t:.1f}s"`

`f"`	Beginning of a *f*ormatted string literal
`Duration =`	string content
`{…}`	content in curly braces is evaluated by Python
`t`	take value from variable t
`:f`	format t as a floating point number
`.1`	display one digit after the decimal point
`s`	string content
`"`	end of formatted string literal

String formatting examples with numbers

```
>>> import math
>>> p = math.pi
>>> f"{p}"  # default representation (same as `str(p)`)
'3.141592653589793'
>>> str(p)
'3.141592653589793'
>>> f"{p:f}"  # as floating point number (6 post-dec digits)
'3.141593'
>>> f"{p:10f}"  # total number 10 characters wide
'  3.141593'
>>> f"{p:10.2f}"  # 10 wide and 2 post-decimal digits
'      3.14'
>>> f"{p:.10f}"  # 10 post-decimal digits
'3.1415926536'
>>> f"{p:e}"  # in exponential notation
'3.141593e+00'
>>> f"{p:g}"  # extra compact
'3.14159'
```

Expressions in f-strings are evaluated at run time

We can evaluate Python expressions in the f-strings:

```
>>> import math
>>> f"The diagonal has length {math.sqrt(2)}."
'The diagonal has length 1.4142135623730951.'
```

---

*Advanced: Precision specifier can also be variables:

```
>>> width = 10
>>> precision = 4
>>> f"{math.pi:{width}.{precision}}"
'     3.142'
```

Show variable name and value with `{name=}`

Convenient short cut for debugging print statements:

```
>>> a = 10
>>> b = 20
>>> c = math.sqrt(a**2 + b**2)
>>> f"State: {a=} {b=} {c=}"
'State: a=10 b=20 c=22.360679774997898'
```

String formatting method overview

“f-strings”: most convenient and recommended method (2016):

```
>>> value = 42
>>> f"the value is {value}"
'the value is 42'
```

“new style” or “advanced” string formatting (Python 3, 2006):

```
>>> "the value is {}".format(value)
'the value is 42'
```

“% operator” (Python 1 and 2):

```
>>> "the value is %s" % value
'the value is 42'
```

Dictionary

Dictionaries

- Python provides another data type: the dictionary.

  Dictionaries are also called “associative arrays” and “hash tables”.

- Dictionaries are *unordered* sets of *key-value pairs*.

  Starting from Python 3.7, dictionaries preserve insertion order.

- An empty dictionary can be created using curly braces:

```
>>> d = {}
```

- Keyword-value pairs can be added like this:

```
>>> d['today'] = '22 deg C'  # 'today' is key
                             # '22 deg C' is value
>>> d['yesterday'] = '19 deg C'
```

Dictionaries

- We can retrieve values by using the keyword as the index:

```
>>> d['today']
'22 deg C'
```

Dictionaries

- `d.keys()` returns all keys:

```
>>> d.keys()
dict_keys(['today', 'yesterday'])
```

- `d.values()` returns all values:

```
>>> d.values()
dict_values(['22 deg C', '19 deg C'])
```

- Check if key is in dictionary:

```
>>> 'today' in d.keys()
True
```

  Equivalent to

Dictionaries

```
>>> 'today' in d
True
```

Dictionary example 1: drinks order

```
order = {}  # create empty dictionary

# add orders as they come in
order['Peter'] = 'Sparkling water'
order['Paul'] = 'Cup of tea'
order['Mary'] = 'Cappuccino'

# deliver order at bar
for person in order.keys():
    print(f"{person} requests {order[person]}")
```

produces this output:

```
Peter requests Sparkling water
Paul requests Cup of tea
Mary requests Cappuccino
```

Iterating over dictionaries

Iterating over the dictionary itself is equivalent to iterating over the keys. Example:

```
order = {}          # create empty dictionary

# add orders as they come in
order['Peter'] = 'Sparkling water'
order['Paul'] = 'Cup of tea'
order['Mary'] = 'Cappuccino'

# iterating over keys:
for person in order.keys():
    print(f"{person} requests {order[person]}")

# is equivalent to iterating over the dictionary:
for person in order:
    print(f"{person} requests {order[person]}")
```

Dictionary example 2: counting objects

```
def count_fruit(fruits):
    """Given a list of fruits (each fruit one string), return a
    dictionary:  each fruit is a key, and the associated value
    reports how often the fruit occurred in the list of fruits.
    """
    d = {}  # start with empty dictionary
    for fruit in fruits:  # process all elements in list fruits
        if fruit not in d:  # this is the first time we find
                            # the fruit in the list
            d[fruit] = 1  # create an entry with key=fruit
        else:  # we have seen this fruit before
            d[fruit] = d[fruit] + 1  # increase counter

    return d

result = count_fruit(['banana', 'apple', 'banana', 'orange'])
print(result)
```

produces this output:

```
{'banana': 2, 'apple': 1, 'orange': 1}
```

Summary dictionaries

- similar to data base
- fast to retrieve value
- useful if you are dealing with two lists at the same time (possibly one of them contains the keyword and the other the value)
- useful if you have a data set that needs to be indexed by strings or tuples (or other immutable objects)
- keys must be immutable (such as strings, numbers, tuples)
- values can be any Python object (including dictionaries)

Default function arguments

Default argument values for functions

- Motivation:
  - suppose we need to compute the area of rectangles and
  - we know the side lengths `a` and `b`.
  - Most of the time, `b=1` but sometimes `b` can take other values.
- Solution 1:

```python
def area(a, b):
    return a * b

print(f"The area is {area(3, 1)}")
print(f"The area is {area(2.5, 1)}")
print(f"The area is {area(2.5, 2)}")
```

Default argument values for functions

- We can make the function more user friendly by providing a *default* value for `b`. We then only have to specify `b` if it is different from this default value:
- Solution 2 (with default value for argument `b`):

```python
def area(a, b=1):
    return a * b

print(f"The area is {area(3)}")
print(f"The area is {area(2.5)}")
print(f"The area is {area(2.5, 2)}")
```

Default argument values for functions

- Default parameters *have to be at the end* of the argument list in the function definition.

Default argument values

You may have met default arguments in use before, for example

- the `print` function uses `end='\n'` as a default value
- the `open` function uses `mode='rt'` as a default value
- the `list.pop` method uses `index=-1` as a default

Keyword function arguments

Keyword argument values

- We can call functions with a “keyword” and a value. (The keyword is the name of the variable in the function definition.)
- Here is an example

```python
def f(a, b, c):
    print(f"{a=} {b=} {c=}")

f(1, 2, 3)
f(c=3, a=1, b=2)
f(1, c=3, b=2)
```

Keyword argument values

which produces this output:

```
    a=1 b=2 c=3
    a=1 b=2 c=3
    a=1 b=2 c=3
```

- If we use *only* keyword arguments in the function call, then we do not need to know the *order* of the arguments. (This is good.)
- Choosing meaningful variable names in the function definition makes the function more user friendly.

*Disallow or enforce keyword argument use

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
      |                |                  |
      |          Positional or keyword    |
      |                                   - Keyword only
      -- Positional only
```

See https://www.python.org/dev/peps/pep-0570/#how-to-teach-this

*Disallow or enforce keyword argument use

```
def standard_arg(arg):
    print(arg)

def pos_only_arg(arg, /):
    print(arg)

def kwd_only_arg(*, arg):
    print(arg)

def combined_example(pos_only, /, standard, *, kwd_only):
    print(pos_only, standard, kwd_only)
```

Virtual Environments `venv`

Virtual environment

Given an installed Python interpreter, we can create virtual environments:

```
python -m venv myvirtualenv
```

and activate that environment (see also next slide):

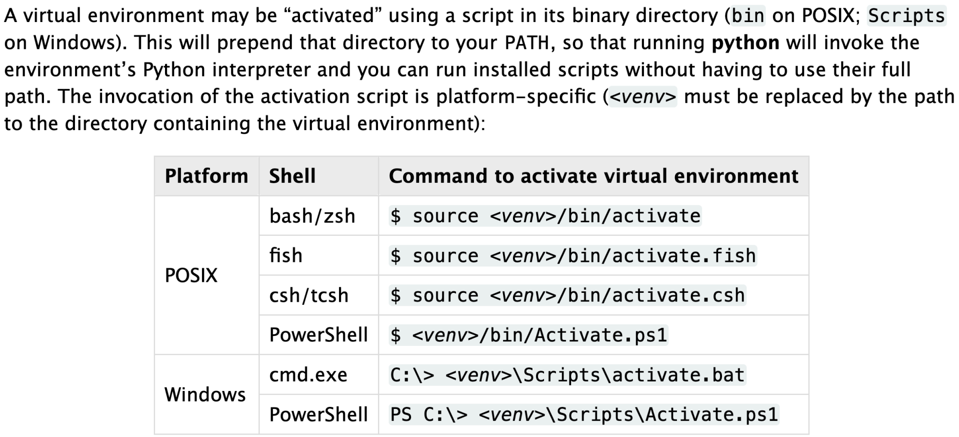
- linux/MacOS: `source myvirtualenv/bin/activate`
- cmd.exe: `myvirtualenv\Scripts\activate.bat`

Why virtual environments?

- good practice: one environment per project
- better reproducibility
- can install two versions of the same library in different environments

Activating virtual environments in different shells

From https://docs.python.org/3/library/venv.html:



A virtual environment may be "activated" using a script in its binary directory (`bin` on POSIX; `Scripts` on Windows). This will prepend that directory to your PATH, so that running **python** will invoke the environment's Python interpreter and you can run installed scripts without having to use their full path. The invocation of the activation script is platform-specific (`<venv>` must be replaced by the path to the directory containing the virtual environment):

Platform	Shell	Command to activate virtual environment
POSIX	bash/zsh	`$ source <venv>/bin/activate`
	fish	`$ source <venv>/bin/activate.fish`
	csh/tcsh	`$ source <venv>/bin/activate.csh`
	PowerShell	`$ <venv>/bin/Activate.ps1`
Windows	cmd.exe	`C:\> <venv>\Scripts\activate.bat`
	PowerShell	`PS C:\> <venv>\Scripts\Activate.ps1`

Installing python packages with `pip`

PyPI

- The Python Package Index (PyPI) provides many python packages (`https://pypi.org`)
- Can search the website for packages, and available versions
- Install locally (in virtual environment) using `pip`

Example: install the python cowsay package:

```
pip install cowsay
```

Uninstall:

```
pip uninstall cowsay
```

pip commands

- `pip install cowsay`
- `pip install cowsay==3.0`
  – install version 3.0
- `pip uninstall cowsay`
- `pip install -U cowsay`
  – upgrade cowsay
- `pip show cowsay`
  - show information about installed package
- `pip list`

Summary virtual environments and pip commands

Summary

- create virtual environment before installing packages
- Common names for virtual environments: `env`, `venv`, `.env`, `.venv`
- use (at least) one virtual environment per project
- use

  ```
  pip freeze
  ```

  and

  ```
  pip install -r requirements.txt
  ```

  to maintain reproducible environments

See more detailed discussion at: https://fangohr.github.io/introduction-to-python-for-computational-science-and-engineering/

*Anaconda and conda packages

- Anaconda software distribution is a convenient way to install python (and more) and python packages
- Anaconda introduced “conda“ packages
- conda packages are not limited to python packages (i.e. more generic)
- If no conda package is available for python package `x`, we can (i) create and activate a conda environment, and (ii) use `pip install x` to install package `x` from the Python Packaging Index (PyPI) via pip

Legal alert: since 2020, anaconda has changed license conditions. If your organisation has more >250 staff, you probably need to pay license fees to use anaconda.

Recommendation:

- use `pixi` (`https://pixi.sh`)
- if you know and like anaconda, use `miniforge` instead (`https://github.com/conda-forge/miniforge`)

*pixi- package management

Pixi is a package and tasks management tool that can install conda and pip packages.

- https://pixi.sh/
- pixi stores its files in the (hidden) subfolder '.pixi'

Example:

```
$ pixi init  # create pixi environment in this folder
$ pixi add python==3.13 numpy  # request python version 3.13
$ pixi add numpy  # add numpy (uses conda-forge package by default)
$ pixi add --pypi cowsay  # add cowsay from PyPI (via pip)
$ pixi shell  # activate pixi environment
<pixi-env> $ python
Python 3.13.0 | packaged by conda-forge | (main, Nov 27 2024, 19:18:26)
>>> import numpy
>>> import cowsay
>>>
```

*For Anaconda users: interaction conda and pip

Anaconda provides packages and (conda) environments through `conda`.

- Avoid mixing pip installs with conda installs, i.e.
    - if conda can install all the required packages, then use that
- if conda cannot install the required package, either
    - first install all that is needed/available from conda
    - then install the desired packages through pip that conda cannot provide
    - afterwards, do not use conda again to install more packages.

  or (if possible)
    - install all packages from pip

See also https://www.anaconda.com/blog/using-pip-in-a-conda-environment

Numpy

numpy

numpy

- is an interface to high performance linear algebra libraries (such as BLAS, LAPACK, ATLAS, MKL, BLIS)
- provides
    - the `array` object (strictly `ndarray` type)
    - fast mathematical operations over arrays
    - linear algebra, Fourier transforms, random number generation
- Numpy is not part of the Python standard library.

numpy 1d-arrays (vectors)

- An (1d) array is a sequence of objects
- all objects in one array are of the same type

```
>>> import numpy as np  # widely used convention
>>> a = np.array([1, 4, 10])  # convert any sequence to array
>>> a
array([ 1,  4, 10])
>>> type(a)
<class numpy.ndarray>
>>> a + 100  # arithmetic operations apply to all elements
array([101, 104, 110])
>>> a**2
array([  1,  16, 100])
>>> np.sqrt(a)
array([ 1.        ,  2.        ,  3.16227766])
>>> a > 3  # apply >3 comparison to all elements
array([False,  True,  True], dtype=bool)
```

Array creation 1: from iterable

- 1d-array (vector) from iterable

```
>>> import numpy as np
>>> a = np.array([1, 4, 10])  # from list
>>> a
array([ 1,  4, 10])
>>> print(a)
[ 1  4 10]
```

- 2d-array (matrix) from nested sequences

```
>>> B = np.array([[0, 1.5], [10, 12]])  # from nested list
>>> B
array([[  0. ,   1.5],
       [ 10. ,  12. ]])
>>> print(B)
[[  0.    1.5]
 [ 10.   12. ]]
```

Array type

- All elements in an array must be of the same type
- For existing array, the type is the `dtype` attribute

```
>>> a.dtype
dtype('int64')
>>> B.dtype
dtype('float64')
```

- We can fix the type of the array when we create the array, for example:

```
>>> a2 = array([1, 4, 10], float)
>>> a2
array([  1.,   4.,  10.])
>>> a2.dtype
dtype('float64')
```

Important array types

- For numerical calculations, we normally use double floats which are known as `float64` or short `float`:

```
>>> a2 = array([1, 4, 10], float)
>>> a2.dtype
dtype('float64')
```

- This is also the default type for `zeros` and `ones`.
- A full list is available at http://docs.scipy.org/doc/numpy/user/basics.types.html

Array size

The `size` of an array is the number of *items*:

```
>>> a.size
3
>>> B.size
4
```

The number of *bytes per item* is the `itemsize`:

```
>>> a.itemsize  # dtype is int64 = 64 bit = 8 byte
8
>>> B.itemsize  # dtype is float64 = 64 bit = 8 byte
8
```

Array size

The total number of bytes of an array is given through the `nbytes` attribute:

```
>>> a.nbytes
24
>>> B.nbytes
32
```

*Diving in with `numpy.info`

```
>>> z = np.arange(0, 12, 1).reshape(3, 4)
>>> z
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> z.dtype
dtype('int64')
>>> np.info(z)
class:  ndarray
shape:  (3, 4)
strides:  (32, 8)  # 32 bytes from row to row
itemsize:  8
aligned:  True
contiguous:  True
fortran:  False
data pointer: 0x6000012dc060
byteorder:  little
byteswap:  False
type: int64
>>> z.nbytes
96
```

Array creation 2: `arange`

- `arange([start,] stop[, step,])` is inspired by `range`: create array from `start` up to *but not including* `stop`

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(10, dtype=float)
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

- `arange` provides non-integer increments:

```
>>> np.arange(0, 0.5, 0.1)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5])
```

Array creation 3: `linspace`

- `linspace(start, stop, num=50)` provides `num` points linearly spaced between `start` and `stop` (*including* `stop`):

```
>>> np.linspace(0, 10, 11)
array([ 0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9., 10.])
>>> np.linspace(0, 1, 11)
array([0.  , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

Array shape

The shape is a tuple that describes

- (i) the dimensionality of the array (that is the length of the shape tuple) and
- (ii) the number of elements for each dimension (“axis”)

Example:

```
>>> a.shape
(3,)    # 1d array with 3 elements
>>> B.shape
(2, 2)  # 2d array with 2 x 2 elements
```

Array shape

Can use shape attribute to change shape:

```
>>> B
array([[  0. ,   1.5],
       [ 10. ,  12. ]])
>>> B.shape
(2, 2)
>>> B.shape = (4,)
>>> B
array([  0. ,   1.5,  10. ,  12. ])
```

Number of dimension also available in attribute `ndim`:

Array shape

```
>>> B.ndim
2
>>> len(B.shape)  # same as B.ndim
2
```

Array indexing (1d arrays)

Regarding indexing, (1d)-Arrays behave like sequences:

```
>>> x = np.arange(0, 10, 2)
>>> x
array([0, 2, 4, 6, 8])
>>> x[3]
6
>>> x[4]
8
>>> x[-1]  # last element
8
```

Array indexing (2d arrays)

```
>>> C = np.arange(12)
>>> C
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
>>> C.shape = (3, 4)
>>> C
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> C[0, 0]  # first index for rows, second for columns
0
>>> C[2, 0]
8
>>> C[2, -1]  # row 3, last column
11
>>> C[-1, -1]  # last row, last column
11
```

Array slicing (1d arrays)

Double colon operator `::`

Read as `START:END:INDEXSTEP`

If either START or END are omitted, the respective ends of the array are used. INDEXSTEP defaults to 1.

Examples:

Array slicing (1d arrays)

```
>>> y = np.arange(10)
>>> y
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> y[0:5]                # slicing (default step is 1)
array([0, 1, 2, 3, 4])
>>> y[0:5:1]              # equivalent (step 1)
array([0, 1, 2, 3, 4])
>>> y[0:5:2]              # slicing with index step 2
array([0, 2, 4])
>>> y[:5:2]               # from the beginning
array([0, 2, 4])
>>> y[0:5:-1]             # negative index step size
array([], dtype=int64)
>>> y[5:0:-1]             # from end to beginning
array([5, 4, 3, 2, 1])
>>> y[5:0:-2]             # in steps of two
array([5, 3, 1])
```

Array slicing (2d)

Slicing for 2d (or higher dimensional arrays) is analog to 1-d slicing, but applied to each component. Common operations include extraction of a particular row or column from a matrix:

```
>>> C
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> C[0, :]           # row with index 0
array([0, 1, 2, 3])
>>> C[:, 1]           # column with index 1
                      # (i.e. 2nd col)
array([1, 5, 9])
```

Array creation 4: `zeros` and `ones`

Other useful methods are `zeros` and `ones` which accept a desired matrix shape as the input:

```
>>> np.zeros((2, 4))   # two rows, 4 cols
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> np.zeros((4,))         # (4,) is tuple
array([ 0.,  0.,  0.,  0.])
>>> np.zeros(4)            # 4 works as well
array([ 0.,  0.,  0.,  0.])

>>> np.ones((2, 7))
array([[ 1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.]])
```

Array creation 5: `eye` and `diag`

Create Identity matrix `eye` (name from capital *I* used in equations):

```
>>> np.eye(2)
array([[1., 0.],
       [0., 1.]])
```

Create diagonal matrix `diag`:

```
>>> np.diag([10, 20, 30])
array([[10,  0,  0],
      [ 0, 20,  0],
      [ 0,  0, 30]])
```

*Views of numpy arrays

Slicing a numpy array results in a *view* of the data (not a copy).

```
>>> C = np.arange(12).reshape(3, 4)
>>> C
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> view_C = C[0, :]
>>> view_C
array([0, 1, 2, 3])
>>> C[0, 0] = 42
>>> view_C
array([42,  1,  2,  3])
```

Often, this is desired — in particular when the arrays are large.

*`array.base` points to the view's data

- `x.base == None` means `x` is not a view.
- `x.base is y` means `x` is a view of `y`.

Example:

```
>>> x = np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> print(x.base)
None
>>> y = x[::2]  # create a view with every 2nd element
>>> print(y.base)
[0 1 2 3 4 5 6 7 8 9]
>>> y.base is x
True
>>> np.shares_memory(x, y)  # do x and y share memory?
True
```

Creating copies of numpy arrays

Create copy of 1d array:

```
>>> y = np.arange(10)
>>> y
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> copy_y = y.copy()
>>> y[0] = 42
>>> copy_y
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> print(copy_y.base)
None
>>> np.shares_memory(y, copy_y)
False
```

Solving linear systems of equations

`np.linealg.solve(A, b)` solves $A\mathbf{x} = \mathbf{b}$ for a square matrix A and given vector $\mathbf{b}$, and returns the solution vector $\mathbf{x}$. Example:

$$A\mathbf{x} = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 4 \end{pmatrix} = \mathbf{b}$$

is equivalent to the system of linear equations:

$$\begin{aligned} 1x_0 + 0x_1 &= 1 \\ 0x_0 + 2x_1 &= 4 \end{aligned}$$

```
>>> A = np.array([[1, 0], [0, 2]])
>>> b = np.array([1, 4])
>>> x = np.linalg.solve(A, b)
>>> x
array([ 1.,  2.])
>>> np.dot(A, x)  # Computing A*x
array([ 1.,  4.])  # this should be b
```

Other linear algebra tools

`help(np.linalg)` provides an overview, including

- `det` to compute the determinant
- `eig` to compute eigenvalues and eigenvectors
- `pinv` to compute the (pseudo) inverse of a matrix
- `svd` to compute a singular value decomposition

Can I always use numpy instead of math?

Use **numpy** instead of **math** so **f** accept scalars (int, float, complex) *and* numpy arrays.

```python
import numpy as np

def f(x):
    """Accepts scalar x or numpy array x and returns exp(-x) * x^2"""
    return np.exp(-x) * x**2

x = 0.5
print(f"Calling with {x=} and {type(x)=}")
print(f"   -> {f(x)=:f} and {type(f(x))=}.")
x = np.array([0.5, 1.0])
print(f"Calling with {x=} and {type(x)=}")
print(f"   -> {f(x)=} and {type(f(x))=}.")
```

Ouput:

```
Calling with x=0.5 and type(x)=<class 'float'>
   -> f(x)=0.151633 and type(f(x))=<class 'numpy.float64'>.
Calling with x=array([0.5, 1. ]) and type(x)=<class 'numpy.ndarray'>
   -> f(x)=array([0.15163266, 0.36787944]) and type(f(x))=<class 'numpy.ndarray'>.
```

Note that for `numpy.exp(x)` for a scalar `x` is slower than `math.exp(x)`.

`numpy` performance optimisation

- numpy is fast if number of elements is large: for an array with one element, `np.sqrt` will be slower than `math.sqrt`
- avoid loops (formulate instead as matrix operation)
- numpy can be up to $\sim$100 times faster than naive Python
- *avoid copies of data (i.e. use views)

arrays are often faster than loops

Without arrays (need to use loop):

```
In [1]: %%timeit
   ...: N = 5000
   ...: mysum1 = 0
   ...: for i in range(N):
   ...:     x = 0.1*i
   ...:     mysum1 += math.sqrt(x)*math.sin(x)
   ...:
657 mu s +- 17.8 mu s per loop (7 runs, 1,000 loops each)
```

Optimised with numpy array:

```
In [2]: %%timeit
   ...: N = 5000
   ...: x = np.arange(0, N)*0.1
   ...: mysum2 = np.sum(np.sqrt(x)*np.sin(x))
   ...:
46.9 mu s +- 19.8 mu s per loop (7 runs, 10,000 loops each)
```

657 μ seconds version 46.9 μ seconds: factor ~ 14

Reading data from text files with `numpy`

```python
import numpy as np

def write_data_file(filename):
    """create test data file with this content:
    0 0
    1 1
    2 4
    3 9
    """
    with open(filename, 'wt') as f:
        for i in range(0, 4):
            f.write(f"{i} {i**2}\n")

write_data_file('test-data.txt')
# read white-space separated data file with numpy.loadtxt:
data = np.loadtxt('test-data.txt')
print(data)
```

Reading data from text files with `numpy`

Ouput:

```
[[0. 0.]
 [1. 1.]
 [2. 4.]
 [3. 9.]]
```

Revisit NOAA data from CSV file (`numpy`)

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

# read data
data = np.loadtxt("data.csv", delimiter=",", skiprows=5)
year = data[:, 0]
dT = data[:, 1]

# plot data
plt.bar(year, dT, color=[0.8, 0, 0])
plt.ylabel("temperature anomaly [deg C]")
plt.xlabel("years")
plt.grid(True)
plt.savefig("anomaly1.pdf")
```

Creates plot on slide 147.

Summary

- numpy provides fast array operations
- elements in the array have the same type (typically a numerical type)
- conversion options include:
    - can create array from sequence `s` with `a = np.array(s)`.
    - can create list from array with `a.tolist()`
- *data is stored contiguously in memory (if possible)

Further reading for numpy

- Consult Numpy documentation if used outside this course. Start here:
  - Basics: `https://numpy.org/doc/stable/user/absolute_beginners.html`
  - Quickstart: `https://numpy.org/doc/stable/user/quickstart.html`
- Matlab users may want to read Numpy for Matlab Users

IPython, Jupyter, Editors and IDEs

IPython (interactive python)

- Interactive Python (`ipython`) prompt
- command history (across sessions), auto completion
- special commands:
  - `%run myfile` will execute file `myfile.py` in current name space
  - `%reset` can delete all objects if required
  - use `range?` instead of `help(range)`
  - `%logstart` will log your session
  - `%prun` will profile code
  - `%timeit` can measure execution time
  - `%load` loads file for editing (also from URL)
  - `%debug` start debugger after error
- Much more (read at http://ipython.org)

Jupyter Notebook useful for research and data science

- Used to be the IPython Notebook, but now supports many more languages (JUlia, PYThon, ER → JUPYTER)
- Notebook is *executable* document hosted in web browser.
- Home page http://jupyter.org

Great value for research

- Fangohr etal: *Data Exploration and Analysis with Jupyter Notebooks* 10.18429/JACoW-ICALEPCS2019-TUCPR02 (2020)
- Granger and Perez: *Thinking and Storytelling with Jupyter*, 10.1109/MCSE.2021.3059263 (2021)
- Fangohr, Di Pierro and Kluyver: *Jupyter in Computational Science*, 10.1109/MCSE.2021.3059494 (2021)
- Beg, Fangohr, etal: *Using Jupyter for reproducible scientific workflows*, Computing in Science and Engineering 23, 36-46 10.1109/MCSE.2021.3052101 (2021)
- Blog entry: Jupyter for Computational Science and Data Science (2022)

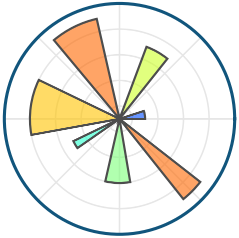
Integrated Development Environments (IDEs) and editors

Including

- Spyder
- PyCharm (commercial)
- Visual studio code
- Emacs
- vim *and* Emacs → Spacemacs
- vim (vi)
- ...

Matplotlib

Matplotlib



- Matplotlib tries to make easy things easy and hard things possible
- Matplotlib is a 2D plotting library which produces publication quality figures (increasingly also 3d)
- Matplotlib can be fully scripted but interactive interface available

Figure and axes windows

- We can have multiple subplots in one figure (`fig`)
- each has one `axes` object (with x-axis and y-axis)
- use `plt.subplots` to create figure and list of axes objects (example next slide)

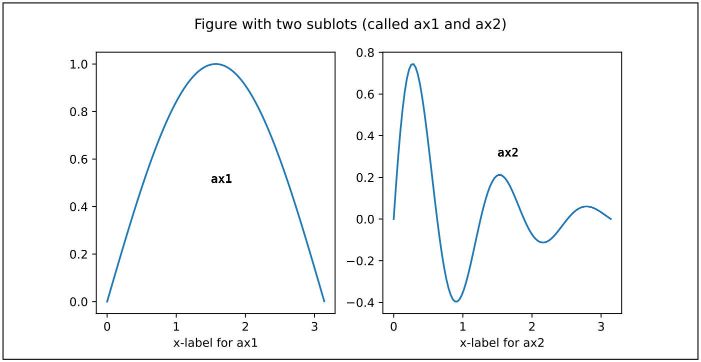
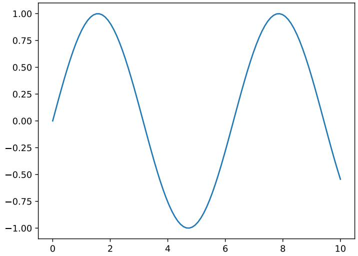


*Figure and axes windows - source

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

x = np.linspace(0, 3.14, 100)
y1 = np.sin(x)
y2 = np.sin(x * 5) * np.exp(-x)

fig, axes = plt.subplots(1, 2, figsize=(8, 4))  # 1 row, 2 cols
ax1, ax2 = axes  # extract the two axes objects
ax1.plot(x, y1)  # plot curve in left subplot
ax1.set_xlabel("x-label for ax1")
ax2.plot(x, y2)  # plot curve in right subplot
ax2.set_xlabel("x-label for ax2")
ax1.text(1.5, 0.5, "ax1", weight="bold", fontfamily="monospace")
ax2.text(1.5, 0.3, "ax2", weight="bold", fontfamily="monospace")
fig.suptitle("Figure with two sublots (called ax1 and ax2)")
fig.savefig("matplotlib-subplot-example.pdf")
```

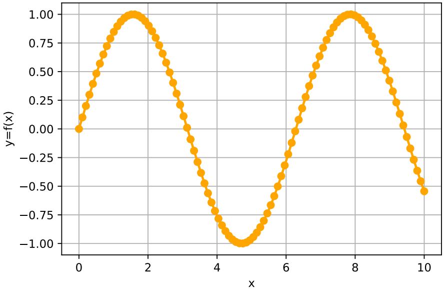
matplotlib.pyplot - example 1

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

xs = np.linspace(0, 10, 100)  # create some data
ys = np.sin(xs)

fig, ax = plt.subplots()  # one figure, one subplot
ax.plot(xs, ys)
fig.savefig("pyplot-demo1.pdf")
```



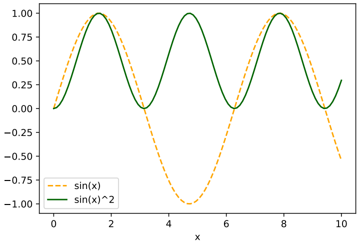
matplotlib.pyplot - example 2: labels and grid

```
fig, ax = plt.subplots(figsize=(6, 4))
ax.plot(xs, ys, 'o-', linewidth=2, color='orange')

ax.grid(True)
ax.set_xlabel('x')
ax.set_ylabel('y=f(x)')
fig.savefig("pyplot-demo2.pdf")
```



matplotlib.pyplot - example 3: two curves

```
xs = np.linspace(0, 10, 100)  # create some data
ys1 = np.sin(xs)
ys2 = np.sin(xs)**2
fig, ax = plt.subplots(figsize=(6, 4))  # plot data
ax.plot(xs, ys1, '--', color='orange', label='sin(x)')
ax.plot(xs, ys2, '-', color='darkgreen', label='sin(x)^2')
ax.set_xlabel('x')
ax.legend()
fig.savefig("pyplot-demo3.pdf")
```



Matplotlib.pyplot interface

- Matplotlib.pyplot is an object oriented plotting interface
- Very fine grained control over plots
- recommended to use

matplotlib.pyplot - references

Matplotlib.pyplot

Matplotlib.pyplot is an object oriented plotting interface.

- prefer this over `pylab`
- Matplotlib tutorials at `https://matplotlib.org/stable/tutorials/index`
- Check gallery at `https://matplotlib.org/stable/gallery/index.html`
- Nicolas Rougier. Scientific Visualization: Python + Matplotlib. Nicolas P. Rougier. 2021, 978-2- 9579901-0-8. hal-03427242, online at `https://github.com/rougier/scientific-visualization-book`

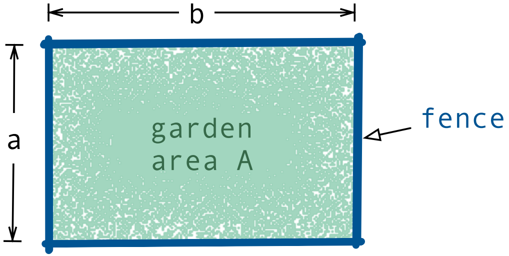
Matplotlib in IPython QTConsole and Notebook

Within the IPython console (for example in Spyder) and the Jupyter Notebook, use

- `%matplotlib inline` to see plots inside the console window, and
- `%matplotlib qt` to create pop-up windows with the plot. (May need to call `matplotlib.show()`.) We can manipulate the view interactively in that window.
- In Spyder, the plots appear by default in the "plots" pane.
- Within the Jupyter notebook, you can use `%matplotlib notebook` which embeds an interactive window in the note book.

Optimisation

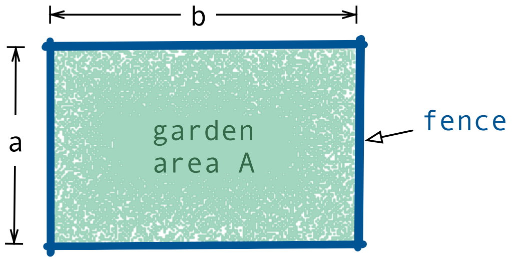
Optimisation example: garden fence



Optimisation problem:

- The shape of the fenced area must be a rectangle (side lengths a and b).
- We have $L = 100\,\mathrm{m}$ of fence available.
- We want to maximise the enclosed garden area $A = ab$.
- What are the optimal values for a and b?

Optimisation example: strategy



How do we find a and b that optimise the area $A(a, b)$?

- We know $L = 100\,\mathrm{m} = 2a + 2b$
- So we have only one unknown: when a is fixed, then b is given by $b = (L - 2a)/2$.
- Change a systematically to find best largest value of A.

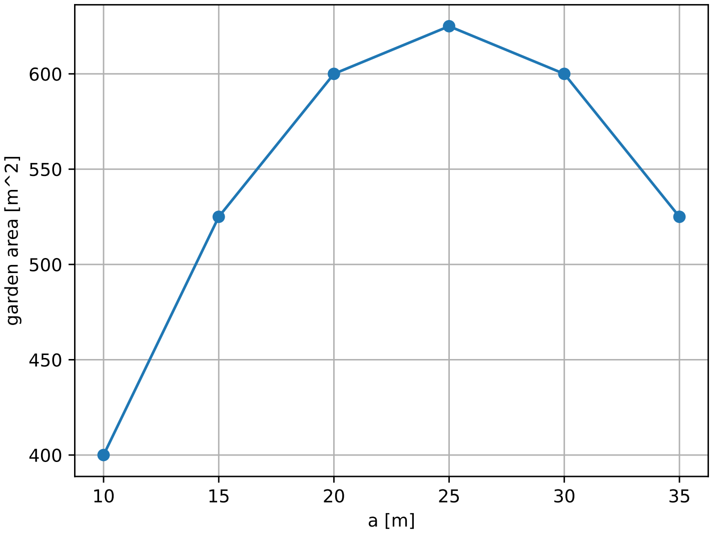
Optimisation example: attempt 1

```
import matplotlib.pyplot as plt



def fenced_area(a):
    """Return area for garden with side length a.

    Given the side length a of a rectangular garden fence
    (with side lengths a and b), compute what side length
    b can be used for a total fence length of 100m.
    Return the associated area.
    """
    L = 100  # total length of fence in metre
    # for a given a, what is length b to use all 100m?
    #    L = 2*a + 2b   =>   b = (L - 2a) / 2
    b = (L - 2*a) / 2
```

Optimisation example: attempt 1

```
# main program
side_lengths = []  # collect the side length a
areas = []  # collect the associated areas

# vary side length of fence a [in metres]
for a in range(10, 40, 5):
    side_lengths.append(a)
    areas.append(fenced_area(a))

plt.plot(side_lengths, areas, '-o')
plt.xlabel('a [m]')
plt.ylabel('garden area [m^2]')
plt.grid(True)
plt.savefig('optimisation-fence.pdf')
```

Optimisation example: attempt 1



Optimisation example: “educational example”

We show one *strategy* to solve an optimisation problem with a simple example so we can focus on the strategy.

For the given fence problem:

- we can guess the correct answer
- there are better ways to find the result with the computer
- we can find the correct answer analytically

Analytical solution

- $A(a) = ab = a\frac{(L-2a)}{2} = \frac{aL}{2} - a^2$
- Find maximum using $\frac{\mathrm{d}A}{\mathrm{d}a} \overset{!}{=} 0 : \frac{\mathrm{d}A}{\mathrm{d}a} = \frac{L}{2} - 2a \Rightarrow a = \frac{L}{4}$
- $b = \frac{L-2a}{2} \Rightarrow b = \frac{L}{4}$
- Check $\frac{\mathrm{d}^2A}{\mathrm{d}a^2} = -2 < 0 \Rightarrow A\left(\frac{L}{4}\right)$ is maximum. ✓

Testing

Testing - context

- Writing software is easy – debugging it is hard
- When debugging, we always *test*
- Later code changes may require repeated testing
- Best to *automate testing* by writing functions that contain tests
- A big topic: here we provide some key ideas
- We use Python extension tool `py.test`, see pytest.org

Example 1: Source code of `mixstrings.py` on following pages.

- a function `mixstrings` is defined together with multiple `test_` functions
- tests are run if `mixstrings.py` is the top-level (tests are not run if file is imported)
- no output if all tests pass (“*no news is good news*”)
- More common approach than calling tests from `__main__`: use `py.test mixstrings.py`

```python
def mixstrings(s1, s2):
    """Given two strings s1 and s2, create and return a new
    string that contains the letters from s1 and s2 mixed:
    i.e. s[0] = s1[0], s[1] = s2[0], s[2] = s1[1],
    s[3] = s2[1], s[4] = s1[2], ...
    If one string is longer than the other, the extra
    characters in the longer string are ignored.

    Example:

    >>> mixstrings("Hello", "12345")
    'H1e2l3l4o5'
    """
    # what length to process
    n = min(len(s1), len(s2))
    # collect chars in this list
    s = []

    for i in range(n):
        s.append(s1[i])
        s.append(s2[i])
    return "".join(s)

```

```python
def test_mixstrings_basics():
    assert mixstrings("hello", "world") == "hweolrllod"
    assert mixstrings("cat", "dog") == "cdaotg"

def test_mixstrings_empty():
    assert mixstrings("", "") == ""

def test_mixstrings_different_length():
    assert mixstrings("12345", "123") == "112233"
    assert mixstrings("", "hello") == ""

if __name__ == "__main__":
    test_mixstrings_basics()
    test_mixstrings_empty()
    test_mixstrings_different_length()
```

`py.test` (also known as `pytest`)

We can use the standalone program `py.test` to run test functions in *any* python program:

- `py.test` will look for functions with names starting with `test_`
- and execute each of those as one test.
- Example:

```
$> py.test -v mixstrings.py
============================= test session starts ===========
platform darwin -- Python 3.10.2, pytest-7.1.2
collected 3 items

mixstrings.py::test_mixstrings_basics PASSED           [ 33%]
mixstrings.py::test_mixstrings_empty PASSED            [ 66%]
mixstrings.py::test_mixstrings_different_length PASSED [100%]
============================== 3 passed in 0.01s ============
```

- This works, even if the file to be tested (here `mixstrings.py`) does not refer to `pytest` at all.

*Calling pytest from a python file

If desired, one can trigger execution of `pytest` from python file.

Example:

```
import pytest

<parts of the file missing here>

if __name__ == "__main__":
    pytest.main(["-v", "mixstrings.py"])
```

However, it is much more common to use `py.test` to discover and execute the tests (often across multiple files).

Testing (partially) defines functionality

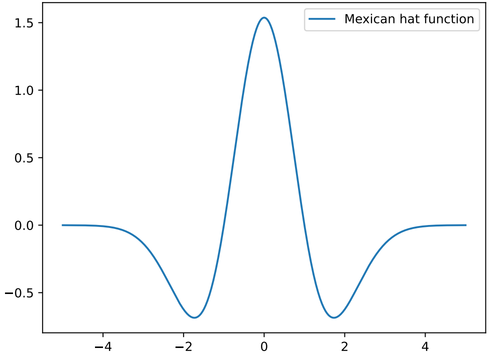
- Just being given the tests for a function, often defines the functions behaviour.
- Example:

```python
def test_reverse_words_empty():
    assert reverse_words("") == ""

def test_reverse_words_one_word():
    assert reverse_words("Python") == "Python"

def test_reverse_words_simple():
    assert reverse_words("Hello world!") == "world! Hello"

def test_reverse_words_with_punctuation():
    assert reverse_words("Hi, there!") == "there! Hi,"
```

Numpy usage examples

Performance gains with numpy

- Calculations using numpy are faster ($\sim$ 100 times) than using pure Python (see example next slide).
- Imagine we need to compute the mexican hat function with many points



Performance gains with numpy

```python
"""Demo: practical use of numpy (mexhat-numpy.py)"""

import datetime
import math
import sys
import time
import matplotlib.pyplot as plt
import numpy as np

N = 100000


def mexhat_py(t, sigma=1):
    """Computes Mexican hat shape, see http://en.wikipedia.org/wiki/Mexican_hat_wavelet
    for equation (13 Dec 2011)"""
    c = 2.0 / math.sqrt(3 * sigma) * math.pi**0.25
    return c * (1 - t**2 / sigma**2) * math.exp(-(t**2) / (2 * sigma**2))


def mexhat_np(t, sigma=1):
    """Computes Mexican hat shape using numpy"""
    c = 2.0 / math.sqrt(3 * sigma) * math.pi**0.25
    return c * (1 - t**2 / sigma**2) * np.exp(-(t**2) / (2 * sigma**2))
```

Performance gains with numpy

```python
def test_is_really_the_same():
    """Checking whether mexhat_np and mexhat_py produce the same results."""
    xs1, ys1 = loop1()
    xs2, ys2 = loop2()
    deviation = math.sqrt(sum((ys1 - ys2) ** 2))
    print("error:", deviation)
    assert deviation < 1e-14


def loop1():
    """Compute list ys with mexican hat function in ys(xs), returns tuple (xs, ys)"""
    xs = np.linspace(-5, 5, N)
    ys = []
    for x in xs:
        ys.append(mexhat_py(x))
    return xs, ys


def loop2():
    """As loop1, but uses numpy to be faster."""
    xs = np.linspace(-5, 5, N)
    return xs, mexhat_np(xs)
```

Performance gains with numpy

```python
def time_this(f):
    """Call f, measure and return number of seconds execution of f() takes"""
    starttime = time.time()
    f()
    stoptime = time.time()
    return stoptime - starttime


def make_plot(filename):
    fig, ax = plt.subplots()
    xs, ys = loop2()
    ax.plot(xs, ys, label="Mexican hat function")
    ax.legend()
    fig.savefig(filename)


def main():
    test_is_really_the_same()
    make_plot("mexhat-numpy.pdf")
    time1 = time_this(loop1)
    time2 = time_this(loop2)
    print(f"Numpy version is {time1 / time2:.1f} times faster")
```

Performance gains with numpy

```python
    print(f"Executed at {datetime.datetime.now()!s} ", end="")
    print(f"with Python {sys.version_info.major}.{sys.version_info.minor}")


if __name__ == "__main__":
    main()
```

Produces this output:

```
error: 1.159820840535702e-15
Numpy version is 81.7 times faster
Executed at 2025-01-19 13:19:42.313469 with Python 3.12
```

- A lot of the source code above is focussed on measuring the execution time.
- Within IPython, we could just have used **`%timeit loop1()`** and **`%timeit loop2()`** to get to the timing information.

0d-arrays with only one item convert to scalars

```
>>> import numpy as np
>>> x = np.array([81., 100.])  # 1d-numpy array with two elements
>>> x.shape
(2,)
>>> np.sqrt(x)
array([ 9., 10.])
>>> math.sqrt(x)  # fails: math.sqrt wants a scalar (e.g. float)
[...]
TypeError: only length-1 arrays can be converted to Python scalars
>>> y = np.array(81.0)  # this is a 0d-numpy array
>>> y.shape
()
>>> math.sqrt(y)        # behaves like a python float
9.0
>>> type(math.sqrt(y))
<class 'float'>
```

0d-arrays with only one item convert to scalars

This allows us to write functions `f(x)` that can take an input argument `x` which can either be a `numpy.array` or a scalar. The `mexhat_np(t)` function is such an example:

```
>>> a = mexhat_np(0); print(f"{a=}")
a=1.537293661343647

>>> a = mexhat_np(np.array([0])); print(f"{a=}")
a=array([1.53729366])

>>> a = mexhat_np(np.linspace(0, 1, 3)); print(f"{a=}")
a=array([1.53729366, 1.01749267, 0.])
```

Pandas

Pandas

- de-facto standard in data science (and maschine learning)
- builds on numpy
- convenient handling of multi-dimensional data sets
- important data structures: `Series` and `DataFrame`
- excellent import and export functionality, including `csv` and `xlsx`.
- many, many, many parameters, functions, tools (Can't know them all)
- for data cleaning and data exploration typically used in Juptyter Notebook

See `https://fangohr.github.io/introduction-to-python-for-computational-science-and-engineering/17-pandas.html`

Revisit NOAA data from CSV file (pandas)

```python
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('data.csv', skiprows=4, index_col=0)
df.plot()  # create line-plot
plt.savefig("anomaly1-pandas-plot.pdf")

# more fine grained control - use matplotlib as usual
plt.close()  # start new plot
year = df.index
dT = df['Anomaly']
plt.bar(year, dT, color=[0.8, 0, 0])
plt.ylabel("temperature anomaly [deg C]")
plt.xlabel("years")
plt.grid(True)
plt.savefig("anomaly1-pandas.pdf")
```

Revisit NOAA data from CSV file (`pandas`)

Creates plot on slide 147.

Common Computational Tasks

Overview common computational tasks

- Data file processing, python, `numpy` & `pandas`
- Data cleaning, data engineering, tabular data (`pandas`)
- Linear algebra fast arrays (`numpy`)
- Random number generation and Fourier transforms (`numpy`)
- Interpolation of data (`scipy.interpolate.interp`)
- Fitting a curve to data (`scipy.optimize.curve_fit`)
- Integrating a function numerically (`scipy.integrate.quad`)
- Integrating a ordinary differential equation numerically (`scipy.integrate.solve_ivp`)

Overview common computational tasks

- Finding the root of a function (`scipy.optimize.fsolve`, `scipy.optimize.brentq`)
- Minimising or maximising a function (`scipy.optimize.fmin`)
- Symbolic manipulation of terms, including integration, differentiation and code generation (`sympy`)

All in the following (third party) python packages:

`scipy`, `numpy`, `pandas`, `sympy`

Optimisation

Optimisation (Minimisation)

- Optimisation typically described as: given a ("objective") function $f(x)$, find x_m so that $f(x_m)$ is the (local) minimum of f.
- Optimisation algorithms need to be given a starting point (initial guess x_0 as close as possible to x_m)
- Minimum position x obtained may be local (not global) minimum

To maximise a function $f(x)$, create a second function $g(x) = -f(x)$ and minimise $g(x)$.

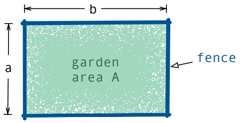
Optimisation example: parabola

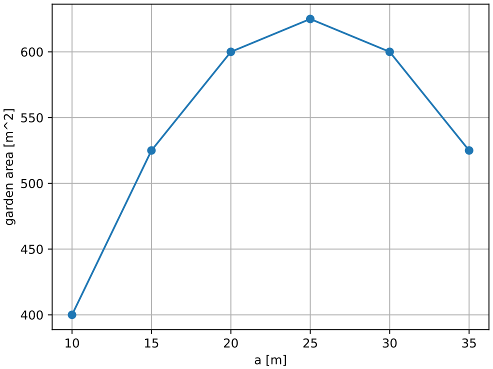
```python
from scipy import optimize

def f(x):
    """parabola - minimum at x=0"""
    return x**2


minimum = optimize.fmin(f, 1)
print("======= Result: ==========")
print(minimum)
```

Code produces this output:

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 17
         Function evaluations: 34
======= Result: ==========
[-8.8817842e-16]
```

Optimisation example: garden fence





Optimisation example: garden fence

```
from scipy.optimize import fmin

def fenced_area(a):
    """Return area for garden with side length a.

    Given the side length a of a rectangular garden fence
    (with side lengths a and b), compute what side length
    b can be used for a total fence length of 100m.
    Return the associated area.
    """
    L = 100  # total length of fence in metre
    # for a given a, what is length b to use all 100m?
    #    L = 2*a + 2b    =>    b = (L - 2a) / 2
    b = (L - 2*a) / 2
    return a*b  # area that fence encloses

def objective_function(a):
    return -1*fenced_area(a)


# main program
a0 = 10  # m, initial guess for fence length of a
a_opt = fmin(objective_function, a0)
print("======= Result: ==========")
print(a_opt)
```

Optimisation example: garden fence

Code produces this output:

```
Optimization terminated successfully.
         Current function value: -625.000000
         Iterations: 22
         Function evaluations: 44
======= Result: ==========
[25.]
```

Optimisation example: multiple minima

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

def f(x):  # objective function
    return np.cos(x) - 3 * np.exp(-((x - 0.2) ** 2))

# find minima of f(x),
# starting from 1.0 and 2.0 respectively
minimum1 = fmin(f, 1.0)
print("Start search at x=1., minimum is", minimum1)
minimum2 = fmin(f, 2.0)
print("Start search at x=2., minimum is", minimum2)

# plot function
x = np.arange(-10, 10, 0.1)
y = f(x)
fig, ax = plt.subplots()
ax.plot(x, y, label=r"$\cos(x)-3e^{-(x-0.2)^2}$")
ax.set_xlabel("$x$")
ax.set_xlabel("$f(x)$")
ax.grid()
ax.axis([-5, 5, -2.2, 0.5])

# add minimum1 to plot
```

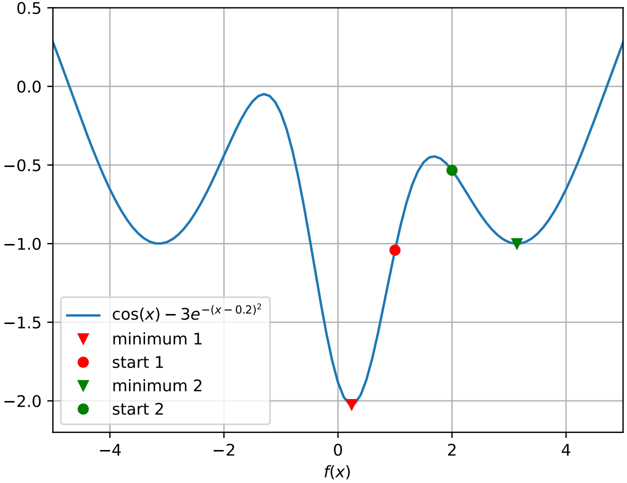
Optimisation example: multiple minima

```python
ax.plot(minimum1, f(minimum1), "vr", label="minimum 1")
# add start1 to plot
ax.plot(1.0, f(1.0), "or", label="start 1")

# add minimum2 to plot
ax.plot(minimum2, f(minimum2), "vg", label="minimum 2")
# add start2 to plot
ax.plot(2.0, f(2.0), "og", label="start 2")

ax.legend(loc="lower left")
fig.savefig("fmin1.pdf")
```

Code produces this output:

```
Optimization terminated successfully.
         Current function value: -2.023866
         Iterations: 16
         Function evaluations: 32
Start search at x=1., minimum is [0.23964844]
Optimization terminated successfully.
         Current function value: -1.000529
         Iterations: 16
         Function evaluations: 32
Start search at x=2., minimum is [3.13847656]
```

Optimisation example: multiple minima



```
commit e77aa60505f23c28a6af95797d85375013d89201
Author: Hans Fangohr <fangohr@users.noreply.github.com>
Date:   Sun Jan 19 11:06:33 2025 +0100

    improve question
```