Python for Computational Science

Hans Fangohr

February 17, 2025

Outline

Outline

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

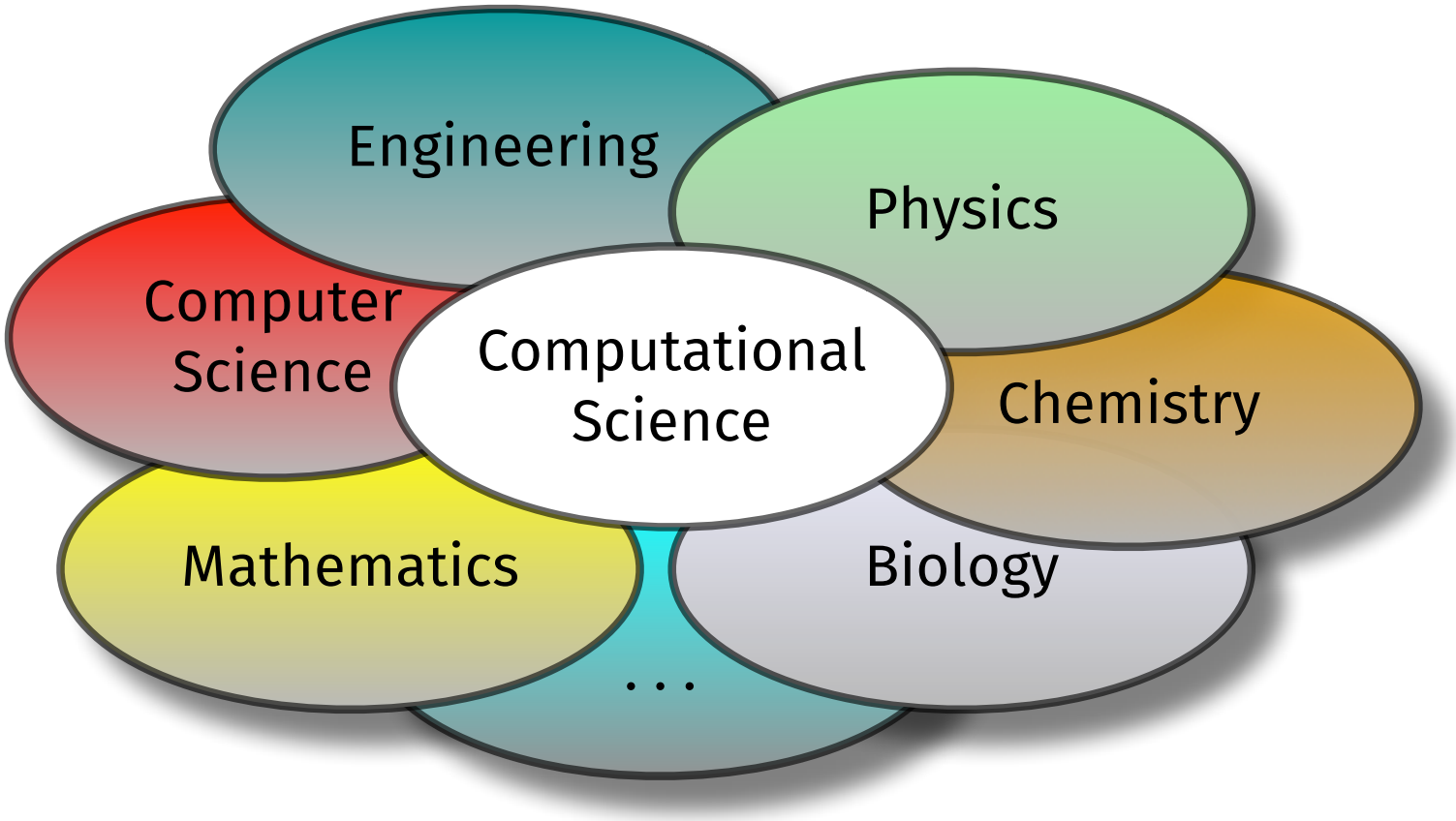
Minimum requirements:

- 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”

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

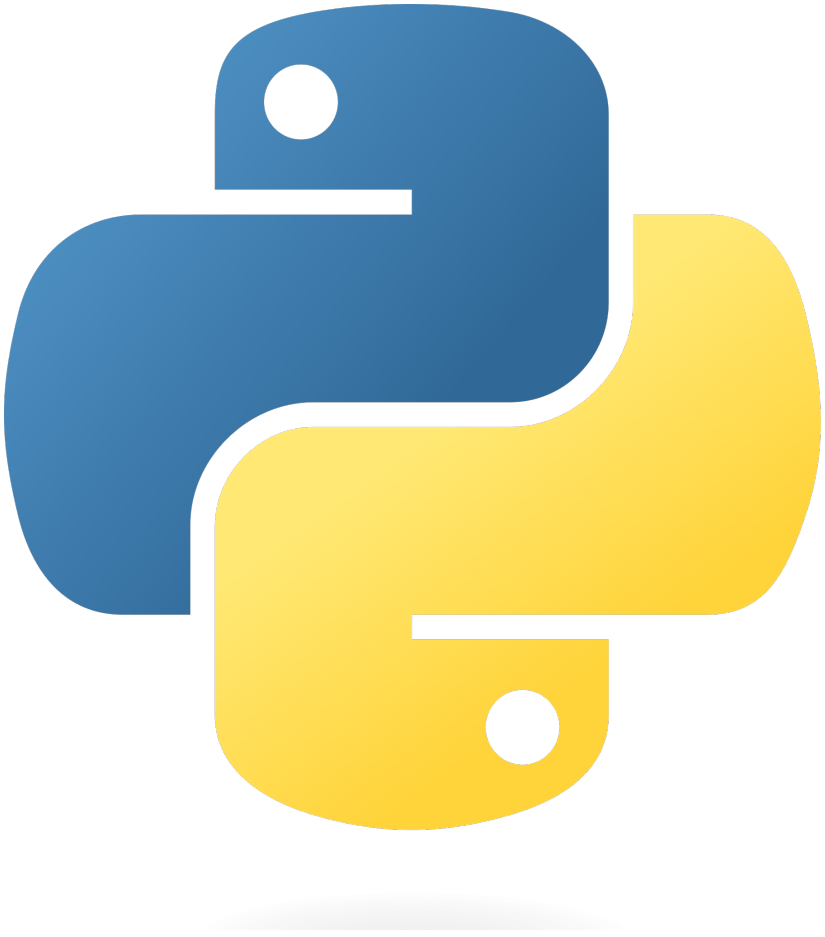
This course: practicalities

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")

```
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

```
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

```
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

```
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 43)
- 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:

```
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:

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

2. use some abbreviation

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

3. import all objects we need explicitly

```
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

```
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:

```
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'
```

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*:

```
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]
```

- 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:

```
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`:

```
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

```
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

```
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)

```
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

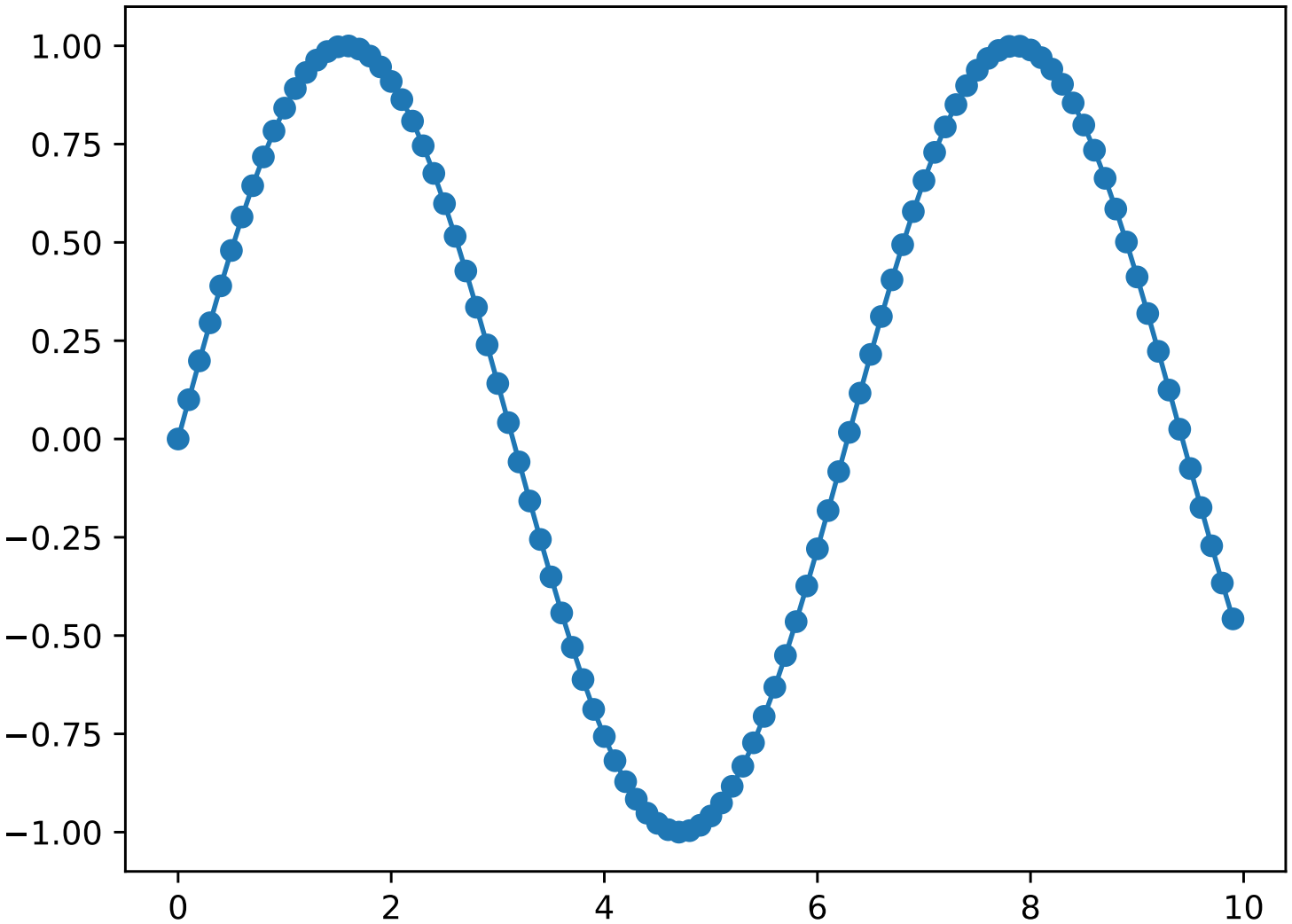
```
import math
import matplotlib.pyplot as plt  # convention

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:

```
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`

```
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

```
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

```
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:

```
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 1/3

```
import matplotlib.pyplot as plt

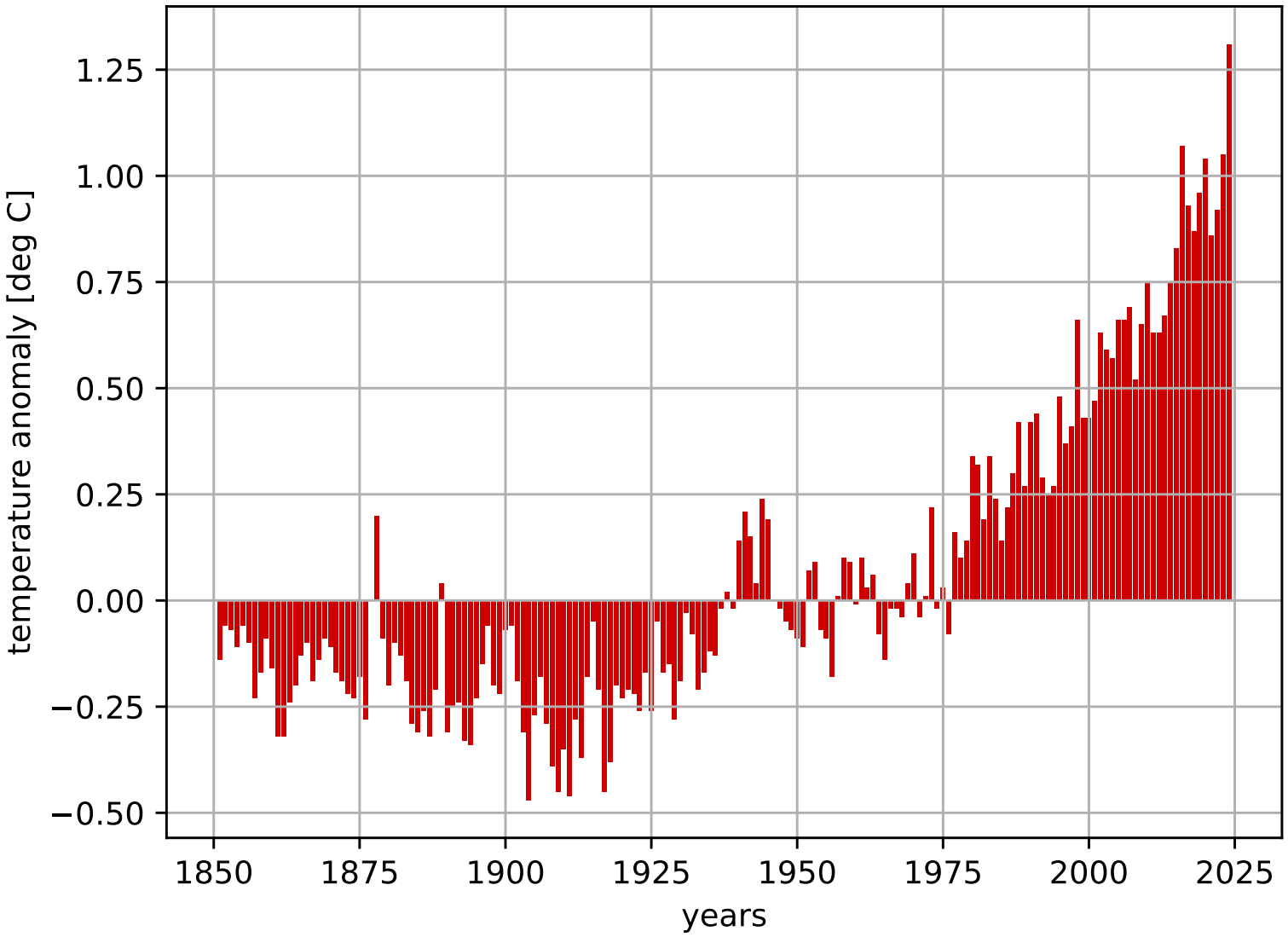
# read data
with open("data.csv", "tr") as f:
    lines = f.readlines()

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
```

```
# 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` (255).

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

Next best is `loadtxt()` from `numpy` (223).

```
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:

```
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:

```
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

```
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:

```
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`):

```
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

```
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)
```

Python installation

Python installation: python interpreter

Options to install the Python interpreter (for example `python 3.12`):

- might be provided by operating system (some version)
- install Python from `https://www.python.org/`
- Linux: use package management of operating system
- MacOS: install via `brew`, (`ports`, `fink`)
- pyenv - `https://github.com/pyenv/pyenv`
- miniforge / miniconda, anaconda
- pixi
- ...

Python packages

Once we have a python interpreter, we can install python packages for that interpreter.

Python packages

- are a set of files
- carry metadata (example: https://github.com/VaasuDevanS/cowsay-python/blob/main/pyproject.toml)
- know which other packages they need ("*dependencies*")
- often in git repository (https://github.com/VaasuDevanS/cowsay-python)
- can be installed with `pip` (see slide 189)
- can be centrally registered (Python Packging Index → PyPI), example: https://pypi.org/project/cowsay/)

Better to use "environments" before installing packages (187)

Tutorial at https://packaging.python.org/en/latest/tutorials/packaging-projects/.

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

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/18-environments.html

Conda packages

- Anaconda software distribution is a convenient way to install python (and more) and python packages
- Anaconda company introduced “conda“ packages, and provides the “anaconda distribution”
- `https://conda-forge.org/` is an open source community-driven effort providing conda-packages (i.e. same format, different provider) and `miniforge`
- Many (but not all) python packages on PyPI have been ported to conda packages
- conda packages are not limited to python packages (i.e. they are more generic)
- conda provides “conda-environments” (similar to virtual environments)
- We can install Python packages via `pip` inside a conda environment.

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

Recommendation:

- if you know and like anaconda (or miniconda), use `miniforge` instead (`https://github.com/conda-forge/miniforge`)
- otherwise use `pixi` (`https://pixi.sh`) (Slide 194)

*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
>>>
```

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.

```
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 ∼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 $\sim$ 14

Reading data from text files with numpy

```
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)

```
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 $\rightarrow$ 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)

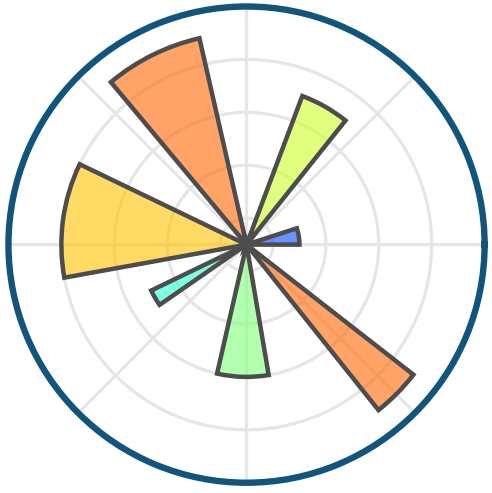
Integrated Development Environments (IDEs) and editors

Including

- Spyder
- PyCharm (commercial)
- Visual studio code
- Emacs
- vim *and* Emacs $\rightarrow$ 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)

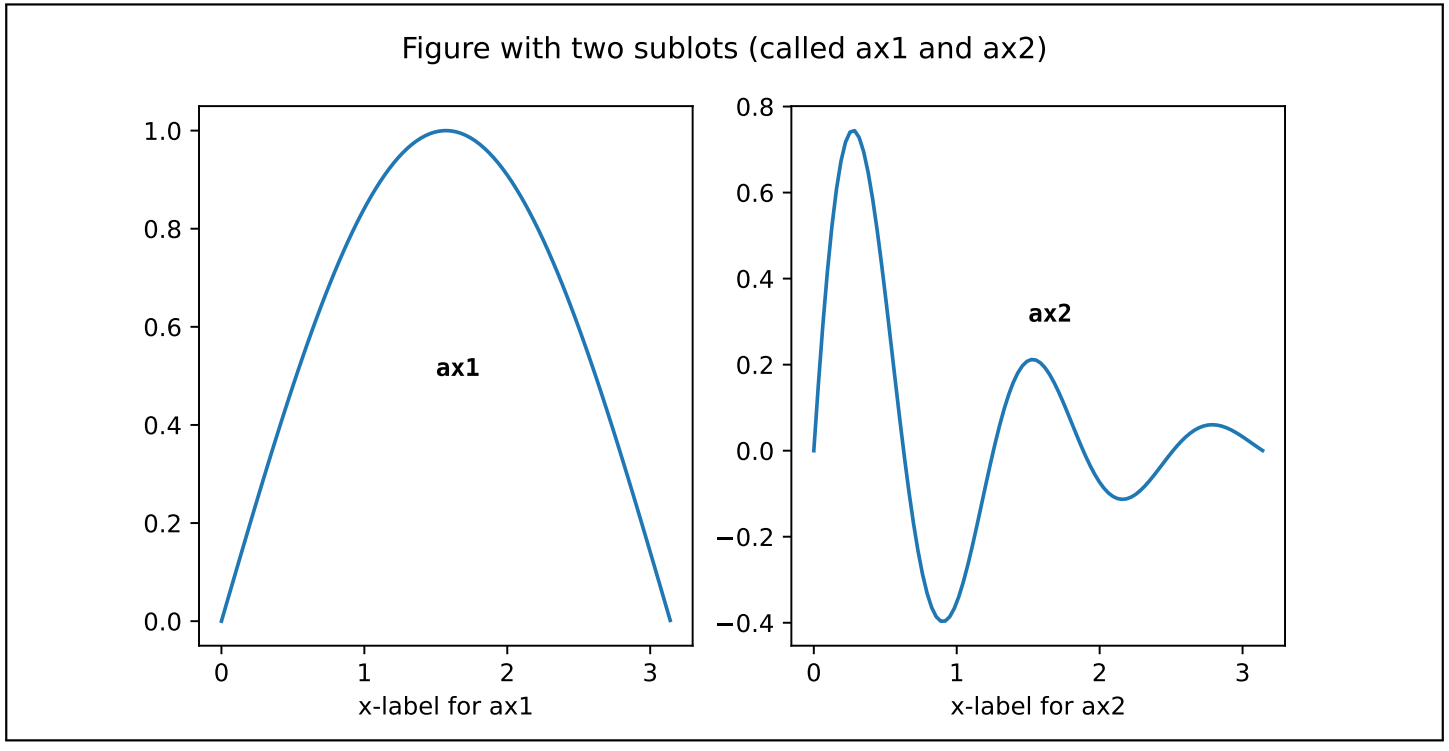


*Figure and axes windows - source

```
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)

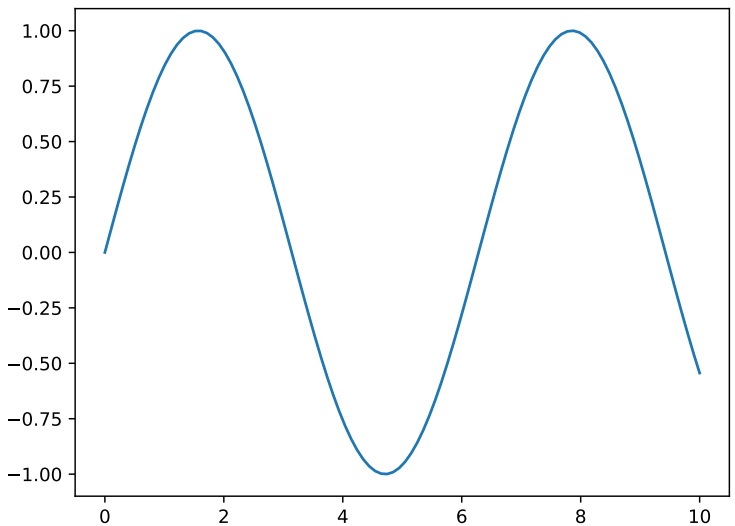
fig, axes = plt.subplots(1, 2, figsize=(8, 4))  # 1 row, 2 cols
ax1, ax2 = axes  # extract the two axes objects
ax1.plot(x, y1)  # plot curve in left subplot
ax1.set_xlabel("x-label for ax1")
ax2.plot(x, y2)  # plot curve in right subplot
ax2.set_xlabel("x-label for ax2")
ax1.text(1.5, 0.5, "ax1", weight="bold", 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

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

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

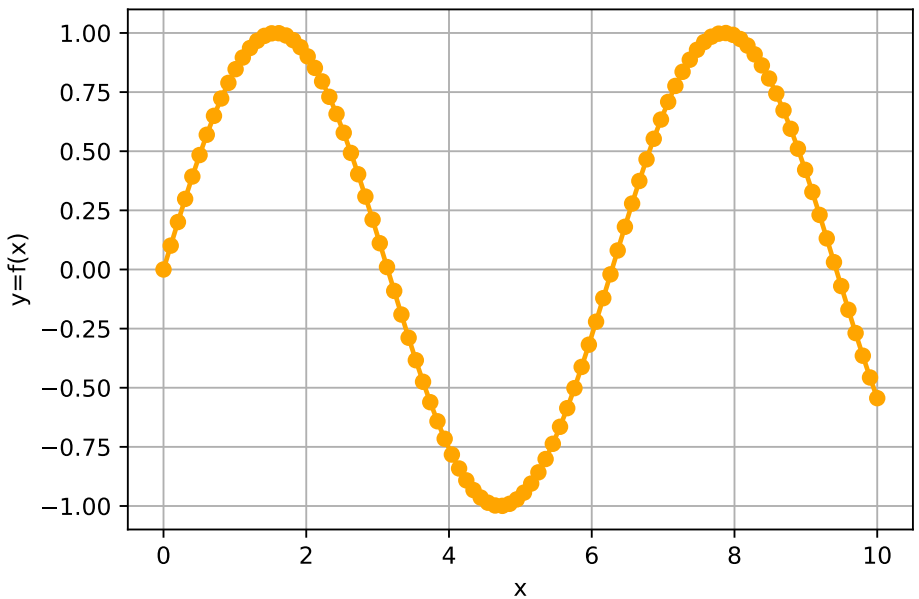
fig, ax = plt.subplots()  # one figure, one subplot
ax.plot(xs, ys)
fig.savefig("pyplot-demo1.pdf")
```



matplotlib.pyplot - example 2: labels and grid

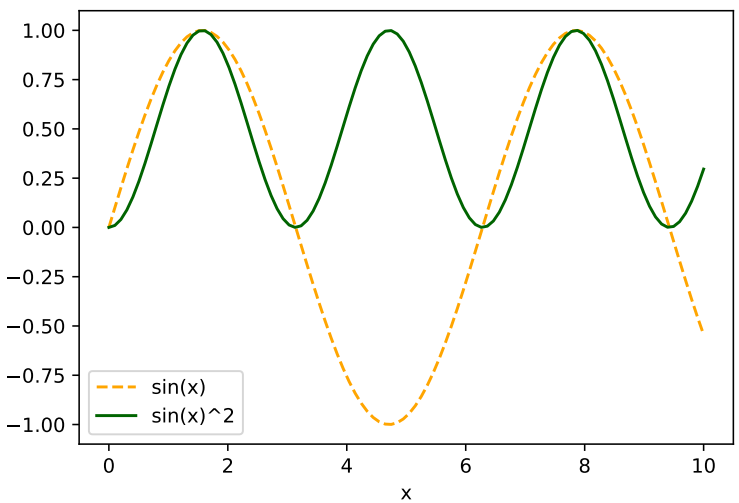
```
fig, ax = plt.subplots(figsize=(6, 4))
ax.plot(xs, ys, 'o-', linewidth=2, color='orange')

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.

- 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 widget` which embeds an interactive window in the note book (needs `ipympl` installed).

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)

```

```
24  def test_mixstrings_basics():
25      assert mixstrings("hello", "world") == "hweolrllod"
26      assert mixstrings("cat", "dog") == "cdaotg"
27
28  def test_mixstrings_empty():
29      assert mixstrings("", "") == ""
30
31  def test_mixstrings_different_length():
32      assert mixstrings("12345", "123") == "112233"
33      assert mixstrings("", "hello") == ""
34
35  if __name__ == "__main__":
36      test_mixstrings_basics()
37      test_mixstrings_empty()
38      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

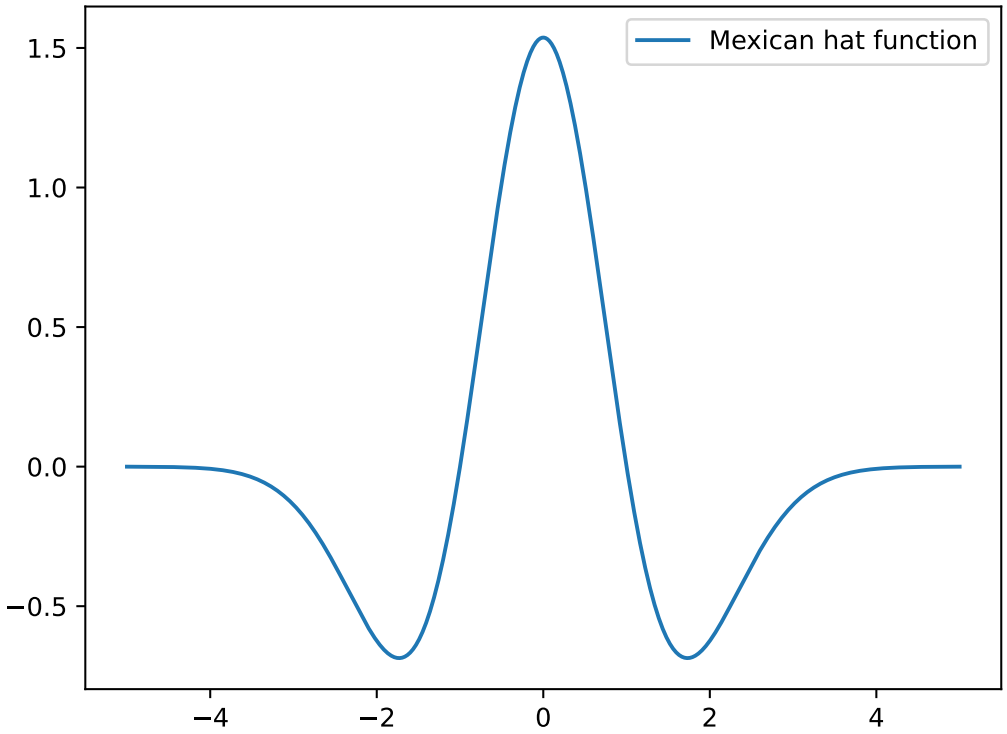
- Just being given the tests for a function, often defines the functions behaviour.
- Example:

```
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

```
"""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

```
26  def test_is_really_the_same():
27      """Checking whether mexhat_np and mexhat_py produce the same results."""
28      xs1, ys1 = loop1()
29      xs2, ys2 = loop2()
30      deviation = math.sqrt(sum((ys1 - ys2) ** 2))
31      print("error:", deviation)
32      assert deviation < 1e-14
33
34
35  def loop1():
36      """Compute list ys with mexican hat function in ys(xs), returns tuple (xs, ys)"""
37      xs = np.linspace(-5, 5, N)
38      ys = []
39      for x in xs:
40          ys.append(mexhat_py(x))
41      return xs, ys
42
43
44  def loop2():
45      """As loop1, but uses numpy to be faster."""
46      xs = np.linspace(-5, 5, N)
47      return xs, mexhat_np(xs)
```

Performance gains with numpy

```
50  def time_this(f):
51      """Call f, measure and return number of seconds execution of f() takes"""
52      starttime = time.time()
53      f()
54      stoptime = time.time()
55      return stoptime - starttime
56
57
58  def make_plot(filename):
59      fig, ax = plt.subplots()
60      xs, ys = loop2()
61      ax.plot(xs, ys, label="Mexican hat function")
62      ax.legend()
63      fig.savefig(filename)
64
65
66  def main():
67      test_is_really_the_same()
68      make_plot("mexhat-numpy.pdf")
69      time1 = time_this(loop1)
70      time2 = time_this(loop2)
71      print(f"Numpy version is {time1 / time2:.1f} times faster")
```

Performance gains with numpy

```
72      print(f"Executed at {datetime.datetime.now()!s} ", end="")
73      print(f"with Python {sys.version_info.major}.{sys.version_info.minor}")
74
75
76  if __name__ == "__main__":
77      main()
```

Produces this output:

```
error: 1.159820840535702e-15
Numpy version is 75.1 times faster
Executed at 2025-02-17 20:04:55.102963 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

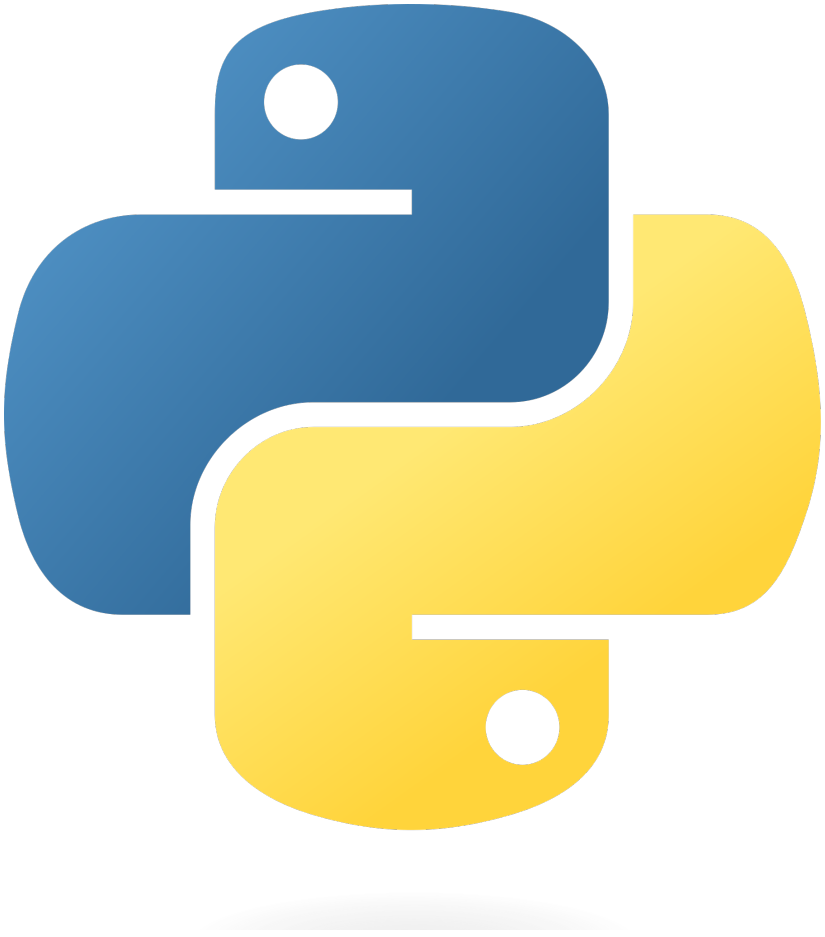
Revisit NOAA data from CSV file (pandas)

```
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.

Part 2



Part 2

Higher Order Functions

Motivational exercise: function tables

- Write a function `print_x2_table()` that prints a table of values of $f(x) = x^2$ for $x = 0, 0.5, 1.0, ..2.5$, i.e.

```
0.0 0.0
0.5 0.25
1.0 1.0
1.5 2.25
2.0 4.0
2.5 6.25
```

- Then do the same for $f(x) = x^3$
- Then do the same for $f(x) = \mathbf{sin}(x)$

Can we avoid code duplication?

Idea: Pass function $f(x)$ to tabulate to tabulating function

Example: (`print_f_table.py`)

```
def print_f_table(f):
    """Given a function f, tabulate it."""
    for i in range(6):
        x = i * 0.5
        print(f"{x} {f(x)}")

def square(x):
    return x ** 2

print_f_table(square)
```

Can we avoid code duplication?

produces

```
0.0 0.0
0.5 0.25
1.0 1.0
1.5 2.25
2.0 4.0
2.5 6.25
```

Can we avoid code duplication (2)?

```
def print_f_table(f):
    for i in range(6):
        x = i * 0.5
        fx = f(x)
        print(f"{x} {fx}")

def square(x):
    return x ** 2

def cubic(x):
    return x ** 3

print("Square"); print_f_table(square)
print("Cubic");  print_f_table(cubic)
```

Can we avoid code duplication (2)?

produces:

```
Square
0.0 0.0
0.5 0.25
1.0 1.0
1.5 2.25
2.0 4.0
2.5 6.25
```

```
Cubic
0.0 0.0
0.5 0.125
1.0 1.0
1.5 3.375
2.0 8.0
2.5 15.625
```

*Example: iterating over functions

- Example (trigtable.py):

```
import math
funcs = [math.sin, math.cos]
for f in funcs:
    fname = f.__name__
    for x in [0, math.pi/2]:
        fx = f(x)
        print(f"{fname}({x:.3f}) = {fx:.3f}")
```

produces

```
    sin(0.000) = 0.000
    sin(1.571) = 1.000
    cos(0.000) = 1.000
    cos(1.571) = 0.000
```

Higher order functions / are first class objects

Functions are 'just' objects in Python. Related terminology:

- Functions are *first class objects* $\leftrightarrow$ functions can be given to other functions as arguments
- Higher order functions accept (or return) functions as arguments.

Iterators

Iterators

- Iterators in Python are useful for efficiently processing sequences of data without loading everything into memory.
- Iterators allow for lazy evaluation, making them ideal for handling large datasets, streaming data
- Iterators can be used to iterator through all elements (of some custom data structure)

Iterators are used in the for loop.

For loop and iterator protocol

For loops can iterate over *iterables*. What are the detailed mechanics?

When we run a for loop, we follow the iterator protocol:

```
>>> for i in iterable:   # iterable could be ['dog', 'cat']
        print(i)
```

- Python calls `iter(iterable)`, which should return an iterator.
- It repeatedly calls `next(iterator)` to retrieve elements.
- When `StopIteration` is raised, the loop stops.

Iterators and iterables

An object `x` is considered an *iterator* if it implements two methods:

- `x.__iter__()`: Returns the iterator object itself
- `x.__next__()`: Returns the next item in the sequence and raises `StopIteration` when there are no more items.

An object is an *iterable* if it can return an iterator using `iter()`.

- Examples for iterables are lists, tuples, dictionaries, sets, and strings.

For-loop example

```
>>> i = iter(["dog", "cat"])      # create iterator
                                  # from list
>>> next(i)
'dog'
>>> next(i)
'cat'
>>> next(i)                       # reached end
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  StopIteration
```

*Generators

- Generators are functions defined using `yield` instead of `return`
- When called, a generator returns an *object that behaves like an iterator*: it has a `__next__()` method.
- Can use generators to *compute* one element sequence at a time (i.e. do not need storage for sequence).

*Generators

```
>>> def squares(n):
...     for i in range(n):
...         yield i**2
...
>>> s = squares(3)
>>> next(s)
0
>>> next(s)
1
>>> next(s)
4
>>> next(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

The execution flow returns at the `yield` keyword (similar to return), but the flow continues after the `yield` when the `next` method is called the next time.

A more detailed example demonstrates this:

*Generators

```
def squares(n):
    print("begin squares()")
    for i in range(n):
        print(f"  before yield i={i}")
        yield i**2
        print(f"  after yield i={i}")

>>> g = squares(3)
>>> next(g)
begin squares()
  before yield i= 0
0
>>> next(g)
  after yield i= 0
  before yield i= 1
1
>>> next(g)
  after yield i= 1
  before yield i= 2
4
>>> next(g)
  after yield i= 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

*Generator for (infinite) sequence of integers

```
def range_infinity():
    """Provides integer numbers starting from 0
    going up to infinity."""
    n = 0
    while True:
        yield n
        n = n + 1


# will run for a long time if not interrupted
for i in range_infinity():
    if i % 1000:  # true every 1000 numbers
        print(i)
```

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)

```

```
24  def test_mixstrings_basics():
25      assert mixstrings("hello", "world") == "hweolrllod"
26      assert mixstrings("cat", "dog") == "cdaotg"
27
28  def test_mixstrings_empty():
29      assert mixstrings("", "") == ""
30
31  def test_mixstrings_different_length():
32      assert mixstrings("12345", "123") == "112233"
33      assert mixstrings("", "hello") == ""
34
35  if __name__ == "__main__":
36      test_mixstrings_basics()
37      test_mixstrings_empty()
38      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).

Advanced Example 2: `factorial.py`

For reference: In this example, we check that an exception is raised if a particular error is made in calling the function.

```
1  import math
2  import pytest
3
4  def factorial(n):
5      """ Compute and return n! recursively.
6      Raise ValueError if n is negative or non-integer.
7
8      >>> from myfactorial import factorial
9      >>> [factorial(n) for n in range(5)]
10     [1, 1, 2, 6, 24]
11     """
12
13     if n < 0:
14         raise ValueError(f"n should be > 0 but n={n}")
15
16     if isinstance(n, int):
17         pass
18     else:
19         raise TypeError(f"n must be integer but is {type(n)}.")
20
21     # actual calculation
22     if n == 0:
23         return 1
24     else:
25         return n * factorial(n - 1)
26
```

```
26
27  def test_basics():
28      assert factorial(0) == 1
29      assert factorial(1) == 1
30      assert factorial(3) == 6
31
32  def test_against_standard_lib():
33      for i in range(20):
34          assert math.factorial(i) == factorial(i)
35
36  def test_negative_number_raises_error():
37      with pytest.raises(ValueError):     # this will pass if
38          factorial(-1)                   # factorial(-1) raises
39                                          # a ValueError
40
41  def test_noninteger_number_raises_error():
42      with pytest.raises(TypeError):
43          factorial(0.5)
```

Output from successful testing:

```
$> py.test -v factorial.py
============================= test session starts ==============
platform darwin -- Python 3.10.2, pytest-7.1.2
collected 4 items

factorial.py::test_basics PASSED                                 [ 25%]
factorial.py::test_against_standard_lib PASSED                   [ 50%]
factorial.py::test_negative_number_raises_error PASSED           [ 75%]
factorial.py::test_noninteger_number_raises_error PASSED         [100%]
============================== 4 passed in 0.02s ===============
```

Notes on pytest

- Normally, we call `py.test` from the command line
- Either give filenames to process (will look for functions starting with `test` in those files)
- or let `py.test` autodiscover all files (!) starting with `test` to be processed.

Example:

```
============================= test session starts ==============
platform darwin -- Python 3.10.2, pytest-7.1.2
collected 7 items

mixstrings.py::test_mixstrings_basics PASSED                [ 14%]
mixstrings.py::test_mixstrings_empty PASSED                 [ 28%]
mixstrings.py::test_mixstrings_different_length PASSED      [ 42%]
factorial.py::test_basics PASSED                            [ 57%]
factorial.py::test_against_standard_lib PASSED              [ 71%]
factorial.py::test_negative_number_raises_error PASSED      [ 85%]
factorial.py::test_noninteger_number_raises_error PASSED    [100%]
============================== 7 passed in 0.01s ===============
```

Testing summary

- Unit testing, integration testing, regression testing, system testing
- absolute key role in modern software engineering: always write (some) tests for your software
- bigger projects have ”continuous integration testing”: automatic execution of tests on any change
- ”eXtreme Programming” (XP) philosophy suggests to write tests *before* you write code (”test-driven-development (TDD)”)

Executable `py.test` and python module `pytest` are not part of the standard python library.

Variables, equality and identity

Variables are references to objects

In Python, variables are references to (or names of) objects.

This is why in the following example, `a` and `b` represent the same list: `a` and `b` are two *different references* to the *same object*:

```
>>> a = [0, 2, 4, 6]  # bind name 'a' to list
>>> a                 # object [0,2,4,6].
[0, 2, 4, 6]
>>> b = a             # bind name 'b' to the same
>>> b                 # list object.
[0, 2, 4, 6]
>>> b[1]              # show second element in list
2                     # object.
>>> b[1] = 10         # modify 2nd elememnt (via b).
>>> b                 # show b.
[0, 10, 4, 6]
>>> a                 # show a.
[0, 10, 4, 6]
```

Identity (`id`, `is`) and equality (`==`)

Identity:

- Two objects `a` and `b` are the *same object* if they live in the same place in memory.
- Python provides the `id` function that returns the *identity* of an object. (It is the memory address.)
- We check with `id(a) == id(b)` wether `a` and `b` are *identical* (i.e. the *same object*).
- `a is b` is equivalent to `id(a) == id(b)`.

Equality:

- Two different objects `a` and `b` can have the *same value*. We check with `a == b` for equality.

Identity (`id`, `is`) and equality (==)

Example 1

```
>>> a = 1
>>> b = 1.0
>>> id(a); id(b)
4298187624          # not in the same place
4298197712          # in memory
>>> a is b          # i.e. not the same objects
False
>>> a == b          # but carry the same value
True
```

Example 2

Identity (`id`, `is`) and equality (`==`)

```
>>> a = [1, 2, 3]
>>> b = a          # b is reference to object of a
>>> a is b         # thus they are the same
True
>>> a == b         # the value is (of course) the same
True
```

Functions – side effect

If we carry out some activity A, and this has an (unexpected) effect on something else, we speak about *side effects*:

```
def sum(xs):
    s = 0
    for i in range(len(xs)):
        s = s + xs.pop()
    return s

xs = [10, 20, 30]
print(f"xs = {xs};   ", end='')
print(f"sum(xs)={sum(xs)};   ", end='')
print(f"xs = {xs}")
```

Output:

```
xs = [10, 20, 30];   sum(xs)=60;   xs = []
```

Functions - side effect 2

Better ways to compute the sum of a list `xs` (or sequence in general)

- use indices to iterate over list

```
def sum(xs):
    s=0
    for i in range(len(xs)):
        s = s + xs[i]
    return s
```

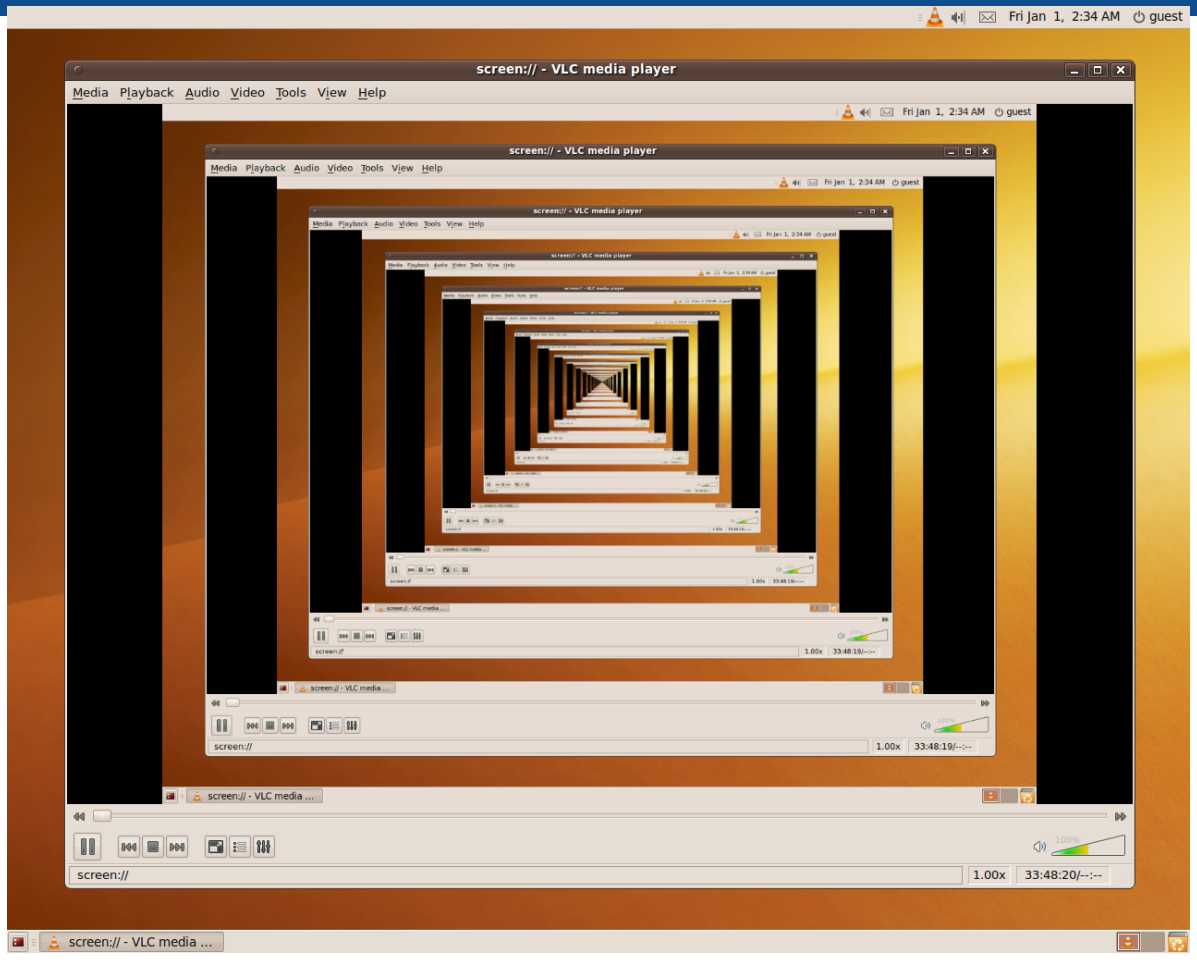
- or (better): iterate over list elements directly

```
def sum(xs):
    s=0
    for elem in xs
        s = s + elem
    return s
```

- or (best) use in-built command `sum(xs)`

Recursion

Recursion



Recursion in a screen recording program, where the smaller window contains a snapshot of the entire screen. Source:

http://en.wikipedia.org/wiki/Recursion

Recursion example: factorial

- Computing the factorial (i.e. $n!$) can be done by computing $(n-1)!n$, i.e. we reduce the problem of size n to a problem of size $n-1$.
- For recursive problems, we always need a *base case*. For the factorial we know that $0! = 1$.
- For $n = 4$:

$$
\begin{aligned}
4! &= 3! \cdot 4 && (1)\\
&= 2! \cdot 3 \cdot 4 && (2)\\
&= 1! \cdot 2 \cdot 3 \cdot 4 && (3)\\
&= 0! \cdot 1 \cdot 2 \cdot 3 \cdot 4 && (4)\\
&= 1 \cdot 1 \cdot 2 \cdot 3 \cdot 4 && (5)\\
&= 24. && (6)
\end{aligned}
$$

Recursion example

Python code to compute the factorial $n! = n * (n-1)!$ recursively:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Usage output:

```
>>> factorial(0)
factorial(0)
1
>>> factorial(2)
2
>>> factorial(4)
24
```

Recursion example Fibonacci numbers

Defined (recursively) as $f(n) = f(n-1) + f(n-2)$ for integers n, and $n > 0$, and $f(1) = 0$ and $f(2) = 1$

Python implementation (fibonacci.py):

```
def f(n):
    if n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return f(n - 2) + f(n - 1)
```

Recursion exercises

1. Write a function `recsum(n)` that sums the numbers from 1 to n *recursively*
2. Study the recursive Fibonacci function from slide 293:
   - what is the largest number n for which we can reasonable compute $f(n)$ within a minute?
   - Can you write faster versions of the Fibonacci function? (There are faster versions with and without recursion.)

str, repr and eval

The `str` function and `__str__` method

All objects in Python should provide a method `__str__` which returns an *informal* string representation of the object.

This method `a.__str__` is called when we apply the `str` function to object `a`:

```
>>> a = 3.14
>>> a.__str__()
'3.14'
>>> str(a)
'3.14'
```

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> now
datetime.datetime(2022, 1, 13, 13, 44, 56, 392268)
>>> str(now)
'2022-01-13 13:44:56.392268'
```

Implicit calling of `str` function

The string method `x.__str__` of object `x` is called implicitly, when we

- pass the object `x` directly to the print command
- use the "`{x}`" notation in f-strings

```
>>> now = datetime.datetime.now()
>>> now
datetime.datetime(2022, 1, 13, 13, 44, 56, 392268)
>>> print(now)
2022-01-13 13:44:56.392268
>>> f"{now}"
'2022-01-13 13:44:56.392268'
```

*The `repr` function and `__repr__` method

- The `repr` function should convert a given object into an *as accurate as possible* string representation
- The `repr` function will generally provide a more detailed string than `str`.
- Applying `repr` to the object `x` will attempt to call `x.__repr__()`.
- The python (and IPython) prompt uses `repr` to 'display' objects.

*The `repr` function and `__repr__` method

Example:

```
>>> import datetime
>>> t = datetime.datetime.now()
>>> str(t)
'2022-01-13 13:55:39.158456'
>>> repr(t)
'datetime.datetime(2022, 1, 13, 13, 55, 39, 158456)'
>>> t
datetime.datetime(2022, 1, 13, 13, 55, 39, 158456)
```

For many objects, `str(x)` and `repr(x)` return the same string.

*The `eval` function

The `eval` function accepts a string, and *evaluates* the string (as if it was entered at the Python prompt):

```
>>> x = 1
>>> eval('x + 1')
2
>>> s = "[10, 20, 30]"
>>> type(s)
<class str>
>>> eval(s)
[10, 20, 30]
>>> type(eval(s))
<class list>
```

*The `repr` and `eval` function

Given an accurate representation of an object as a string, we can convert that string into the object using the `eval` function.

```
>>> i = 42
>>> type(i)
<class int>
>>> repr(i)
'42'
>>> type(repr(i))
<class str>
>>> eval(repr(i))
42
>>> type(eval(repr(i)))
<class int>
```

*The `repr` and `eval` function

The `datetime` example:

```
>>> import datetime
>>> t = datetime.datetime.now()
>>> t_as_string = repr(t)
>>> t_as_string
'datetime.datetime(2016, 9, 8, 14, 28, 48, 648192)'
>>> t2 = eval(t_as_string)
>>> t2
datetime.datetime(2016, 9, 8, 14, 28, 48, 648192)
>>> type(t2)
<class datetime.datetime>
>>> t == t2
True
```

List comprehension

List comprehension - one slide summary

```
>>> xs = [2*i for i in range(5)]  # 'list comprehension'
>>> print(xs)
[0, 2, 4, 6, 8]
```

is equivalent to this for set of commands with a for loop:

```
>>> xs = []
>>> for i in range(5):
...     xs.append(2*i)
...
>>> print(xs)
[0, 2, 4, 6, 8]
```

- useful when we need to process or create a list quickly
- no additional functionality over for-loop
- sometimes more elegant (≈ shorter) than for-loop

List comprehension

- List comprehension follows the mathematical "set builder notation"
- Convenient way to process a list into another list (without for-loop).

Examples

```
>>> [2*i for i in range(5)]
[0, 2, 4, 6, 8]

>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List comprehension structure

Structure of list comprehension:

```
[EXPRESSION(OBJECT) for OBJECT in ITERABLE]
```

where `EXPRESSION`, `OBJECT`, and `ITERABLE` can vary.

Examples:

```
>>> [2*i for i in range(5)]
[0, 2, 4, 6, 8]

>>> import math
>>> [math.sqrt(x) for x in [1, 4, 9, 16]]
[1.0, 2.0, 3.0, 4.0]

>>> [s.capitalize() for s in ["dog", "cat", "mouse"]]
['Dog', 'Cat', 'Mouse']
```

List comprehension example 1 and 2

Can be useful to populate lists with numbers quickly

- Example 1:

```
>>> ys = [x**2 for x in range(10)]
>>> ys
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Example 2:

```
>>> import math
>>> xs = [0.1 * i for i in range(5)]
>>> xs
[0.0, 0.1, 0.2, 0.3, 0.4]
>>> ys = [math.exp(x) for x in xs]
>>> ys
[1.0, 1.1051709180756477, 1.2214027581601699,
 1.3498588075760032, 1.4918246976412703]
```

List comprehension with filter

```
[EXPRESSION(OBJECT) for OBJECT in ITERABLE
                        if CONDITION(OBJECT)]
```

- include `OBJECT` only if `CONDITION(OBJECT)` is `True`.
- Example:

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

>>> [i for i in range(10) if i > 5]
[6, 7, 8, 9]

>>> [i for i in range(10) if i**2 > 5]
[3, 4, 5, 6, 7, 8, 9]
```

*Dictionary comprehension

In addition to *list comprehension* there is also *dictionary comprehension* available:

```
>>> {x: x**2 for x in range(5)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

>>> {word: len(word) for word in ["dog", "bird", "mouse"]}
{'dog': 3, 'bird': 4, 'mouse': 5}
```

The structure is

```
{KEY(OBJECT) : VALUE(OBJECT) for OBJECT in ITERABLE}
```

*Generator comprehension (advanced)

Generators (see slide 269) can also be created using a comprehension syntax:

```
>>> gen = (x**2 for x in range(5))
>>> type(gen)
<class 'generator'>
>>> for item in gen:
...     print(item)
...
0
1
4
9
16
>>> list( (x**2 for x in range(5)) )
[0, 1, 4, 9, 16]
>>>
```

Object Oriented (OO) Programming

Overview

- Motivation and terminology
- Time example
  - encapsulation
  - defined interfaces to hide data and implementation
  - operator overloading
  - inheritance
  - (teaching example only: normally `datetime` and others)
- Geometry example
- Objects we have used already
- Summary

Motivation

- When programming we often store *data*
- and *do* something with the data.
- For example,
  - an array keeps the data and
  - a function does something with it.
- Programming driven by actions (*i.e.* calling functions to do things) is called *imperative* or *procedural* programming.

Object Orientation

- merge data and functions (that operate on this data) together into *classes*.

(...and objects are “instances of a class”)

Terminology

- a class combines data and functions
  (think of a class as a blue print for an object)
- objects are *instances* of a class
  (you can build several objects from the same blue print)
- a class contains *members*
- members of classes that store data are called *attributes*
- members of classes that are functions are called *methods* (or behaviours)

Example 1: a class to deal with time

```
class Time:
    def __init__(self, hour, min):
        self.hour = hour
        self.min = min

    def print24h(self):
        print(f"{self.hour:2}:{self.min:2}")

    def print12h(self):
        if self.hour < 12:
            ampm = "am"
        else:
            ampm = "pm"

        print(f"{self.hour % 12:2}:{self.min:2} {ampm}")
```

Example 1: a class to deal with time

```
if __name__ == "__main__":
    t = Time(15, 45)

    print("print as 24h: "),
    t.print24h()
    print("print as 12h: "),
    t.print12h()

    print(f"The time is {t.hour} hours and {t.min} minutes.")
```

Produces this output:

```
print as 24h:
15:45
print as 12h:
 3:45 pm
The time is 15 hours and 45 minutes.
```

- `class Time:` starts the definition of a class with name `Time`
- `__init__` is the *constructor* and is called whenever a new object is initialised
- all methods in a class need `self` as the first argument. Self represents the object.
- variables can be stored and are available everywhere *within* the object when assigned to `self`, such as `self.hour` in the example.
- in the main program:
  - `t = Time(15, 45)` creates the object `t`
    $\leftrightarrow$ `t` is an instance of the class `Time`
  - methods of `t` can be called like this `t.print24h()`.

This was a mini-example demonstrating how data attributes (*i.e.* `hour` and `min`) and methods (*i.e.* `print24h()` and `print12h()`) are combined in the `Time` class.

Members of an object

- In Python, we can use `dir(t)` to see the members of an object `t`. For example:

```
>>> t = Time(15, 45)
>>> dir(t)
['__class__', '__doc__', ...<entries removed here>....,
  'hour', 'min', 'print12h', 'print24h']
```

- We can also modify attributes of an object using for example `t.hour = 10`. However, *direct* access to attributes is sometimes supressed (although it may look like direct access $\rightarrow$ property).

Data hiding (also: information hiding)

- A well designed class provides methods to get and set attributes.
- These methods define the *interface* to that class.
- Purpose of get and set methods:
  - to perform error and consistency checking when values are set
  - to hide the implementation of the class (from the user):
    - we can change the implementation of the class without changing the interface (and a user of the class would never know)
    - makes future changes possible
- We introduce set and get methods as one would use in Java and C++ to reflect the common ground in OO class design. *In Python, the use of* `property` *is often preferred over set and get methods.*

Example 2: a class to deal with time

```
1   class Time:
2       def __init__(self, hour, min):
3           self.setHour(hour)
4           self.setMin(min)
5
6       def setHour(self, hour):
7           if 0 <= hour <= 23:
8               self._hour = hour
9           else:
10              raise ValueError(f"Invalid hour value: {hour}")
11
12      def setMin(self, min):
13          if 0 <= min <= 59:
14              self._min = min
15          else:
16              raise ValueError(f"Invalid min value: {min}")
```

Example 2: a class to deal with time

```
18      def getHour(self):
19          return self._hour
20
21      def getMin(self):
22          return self._min
23
24      def print24h(self):
25          print(f"{self.getHour():2}:{self.getMin():02}")
26
27      def print12h(self):
28          if self._hour < 12:
29              ampm = "am"
30          else:
31              ampm = "pm"
32
33          print(f"{self._hour%12:2}:{self._min:2} {ampm}")
34
```

Example 2: a class to deal with time

```
36  if __name__ == "__main__":
37      t = Time(15, 45)
38
39      print("print as 24h: "),
40      t.print24h()
41      print("print as 12h: "),
42      t.print12h()
43      print(f"that is {t.getHour()} hours and {t.getMin()} minutes")
```

which produces

```
print as 24h:
15:45
print as 12h:
 3:45 pm
that is 15 hours and 45 minutes
```

Data Hiding summary

- providing *set* and *get* methods for attributes of an object
- The pythonic way for get and set functions is through *properties*. A property is a special attribute:
  - get and set functions are called automatically when the attribute is accessed or assigned to.
  - add these lines to create the properties `min` and `hour`:

```
hour = property(fget=getHour, fset=setHour)
min = property(fget=getMin, fset=setMin)
```

*Private members

- Advanced: Attributes and methods that the user cannot access directly are called *private*.
  - In Python, class members can never be truly private. (in contrast to C++, Java, ...)
  - Convention: an attribute starting with an underscore is private, and should not be accessed directly (by the user of the class). Example: `self._hour`

Operator overloading

- We constantly use operators to “do stuff” with objects.
- What the operator does, depends on the objects it operates on. For example:

```
>>> a = "Hello "; b = "World"
>>> a + b                    # concatenation
'Hello World'
>>> c = 10; d = 20
>>> c + d                    # addition
30
```

- This is called *operator overloading* because the operation is overloaded with more than one meaning.
- Other operators include `-`,`*` , `**`, `[]`, `()`, `>`, `>=`, `==`, `<=`, `<`, `str()`, `repr()`, ...
- We can overload these operators for our own objects. The next slide shows an example that overloads the `>` operator for the `Time` class.
- It also overloads the “str” and “repr“ functions.

```python
class Time:
    def __init__(self, hour, min):
        self.hour, self.min = hour, min

    def __str__(self):
        """overloading the str operator (STRing)"""
        return f"[ {self.hour:2d}:{self.min:2d} ]"

    def __repr__(self):
        """overloading the repr operator (REPResentation)"""
        return f"Time({self.hour:2d}, {self.min:2d})"

    def __gt__(self, other):
        """overloading the GreaterThan operator"""
        selfminutes = self.hour * 60 + self.min
        otherminutes = other.hour * 60 + other.min
        return selfminutes > otherminutes
```

```
if __name__ == "__main__":
    t1 = Time(15, 45)
    t2 = Time(10, 55)

    print(f"Informal string representation of t1: {str(t1)}")
    print(f"Representation of object = {repr(t1)}")

    print("compare t1 and t2: "),
    if t1 > t2:
        print("t1 is greater than t2")
```

Output:

```
Informal string representation of t1: [ 15:45 ]
Representation of object = Time(15, 45)
compare t1 and t2:
t1 is greater than t2
```

Inheritance

- Sometimes, we need classes that share certain (or very many, or all) attributes but are slightly different.
- Example 1: Geometry
  - a point (in 2 dimensions) has an *x* and *y* attribute
  - a circle is a point with a radius
  - a cylinder is a circle with a height
- Example 2: People at universities
  - A person has an address.
  - A student is a person and selects modules.
  - A lecturer is a person with teaching duties.
  - ...
- In these cases, we can define a *base class* (or *parent class*) and *derive* other classes from it.
- This is called *inheritance*

Inheritance example Time

```
class Time:
    def __init__(self, hour, min):
        self.hour = hour
        self.min = min

    def __str__(self):
        """overloading the str operator (STRing)"""
        return f"[ {self.hour:2}:{self.min:02} ]"

    def __gt__(self, other):
        """overloading the GreaterThan operator"""
        selfminutes = self.hour * 60 + self.min
        otherminutes = other.hour * 60 + other.min
        return selfminutes > otherminutes
```

Inheritance example Time

```
class TimeUK(Time):
    """Derived (or inherited class)"""
    def __str__(self):
        """overloading the str operator (STRing)"""
        if self.hour < 12:
            ampm = "am"
        else:
            ampm = "pm"

        return f"[ {self.hour%12:2}:{self.min:02} {ampm}]"

if __name__ == "__main__":
    t3 = TimeUK(15, 45)
    t4 = Time(16, 15)
    print(t3)
    print(t4)

    if t3 > t4:
        print("t3 is greater than t4")
    else:
        print("t3 is not greater than t4")
```

Inheritance example Time

Output:

```
[  3:45 pm]
[ 16:15 ]
t3 is not greater than t4
```

*Inheritance example Geometry

```
import math


class Point:  # this is the base class
    """Class that represents a point"""

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y


class Circle(Point):  # is derived from Point
    """Class that represents a circle"""

    def __init__(self, x=0, y=0, radius=0):
        Point.__init__(self, x, y)
        self.radius = radius

    def area(self):
        return math.pi * self.radius**2
```

*Inheritance example Geometry

```
class Cylinder(Circle):  # is derived from Circle
    """Class that represents a cylinder"""

    def __init__(self, x=0, y=0, radius=0, height=0):
        Circle.__init__(self, x, y, radius)
        self.height = height

    def volume(self):
        return self.area() * self.height


if __name__ == "__main__":
    d = Circle(x=0, y=0, radius=1)
    print("circle area:", d.area())
    print("attributes of circle object are")
    print([name for name in dir(d) if name[:2] != "__"])
    c = Cylinder(x=0, y=0, radius=1, height=2)
    print("cylinder volume:", c.volume())
    print("attributes of cylinder object are")
    print([name for name in dir(c) if name[:2] != "__"])
```

*Inheritance example Geometry

Output:

```
circle area: 3.141592653589793
attributes of circle object are
['area', 'radius', 'x', 'y']
cylinder volume: 6.283185307179586
attributes of cylinder object are
['area', 'height', 'radius', 'volume', 'x', 'y']
```

*Inheritance (2)

- if class A should be derived from class B we need to use this syntax:

  `class A(B):`
- Can call constructor of base class explicitly if necessary (such as in `Circle` calling of `Point.__init__(...)`)
- Derived classes inherit attributes and methods from base class (see output on previous slide: for example the cylinder and circle object have inherited `x` and `y` from the point class).

*super()

In the Circle class definition, we can replace

```
Point.__init__(self, x, y)
```

with

```
super().__init__(x, y)
```

as a short cut to call a method from the (single) parent class.

(Same for the Cylinder class definition.)

Everything in Python is an object

All “things” in Python are objects, including numbers, strings and functions.

```
>>> dir(42)              # numbers are objects
>>> dir(list)            # list is an object
>>> import math
>>> dir(math)            # modules are objects
>>> dir(lambda x: x)     # functions are objects
```

Summary Object Oriented Programming

Summary

- Object orientation is about merging data and functions into one object (sometimes called encapsulation).
- Data hiding (through get and set methods) makes the classes more flexible: easier to maintain, possible to change internal implementation
- Through operator overloading we can make working with the objects more convenient and more flexible
- Classes can be derived from other classes: facilitates re-use of code

Typing

Dynamic Typing

Python derives flexibility from being dynamically typed:

```
def add(x, y):
    """Type of x and y is dynamic."""
    print(f"Type of {x=} is {type(x)}")
    return x + y

print(add(10, 20))
print(add("Hello", " World"))
```

Output:

```
Type of x=10 is <class 'int'>
30
Type of x='Hello' is <class 'str'>
Hello World
```

Duck typing — behaviour more important than type

```
def print_length(x):
    """Works for every object with __len__ method."""
    print(f"The object of type {type(x)} has length {len(x)}.")

class Len42class:
    """A class where every object has length 42."""
    def __len__(self):
        return 42

x = [10, 20]
print_length(x)  # list has length
y = Len42class()  # y has length
print_length(y)
```

Output:

```
The object of type <class 'list'> has length 2.
The object of type <class '__main__.Len42class'> has length 42.
```

Static typing

- More formal “static typing” information can be useful:
  - better (machine readable) documentation of types
  - static type checking may discover mistakes
  - editors/IDEs can use static type information
  - potential execution speed-up (see cython)
- Typing module for type annotation introduced in Python 3.5
- Relevant PEPs: PEP483 and PEP484
- More concise introduction to typing realpython.com

Type annotation example

- Function type annotation: expect `str` and return `str`

```
1  def hello(name: str) -> str:
2      """Given a name, return 'Hello ' + name."""
3      return "Hello " + name
4
5  hello("Paul")  # correct function call
6  hello(42)  # incorrect type
```

- Can use `mypy` to do static type analysis:

```
typing-static1.py:6: error: Argument 1 to "hello" has
↪  incompatible type "int"; expected "str"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

Gradual typing

- *gradual* introduction of type annotations is possible: can introduce type annotation for some functions only
- effective to annotate most heavily used functions first
  - they are called from other places
  - accidental calls with incorrect types can be discovered

Gradual typing example

```
1  def mysum(a: int, b: int) -> int:
2      """Expect two ints and return the sum."""
3      return a + b
4
5  def f_without_types(x):
6      """Return x. A function without type annotation."""
7      return x
8
9  print(mysum(2, 3))
10 print(mysum("Hello", 2023))  # will not work
```

- Can use `mypy` to do static type analysis:

```
typing-gradual.py:10: error: Argument 1 to "mysum" has
↪  incompatible type "str"; expected "int"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

Type annotation summary

Typing in Python

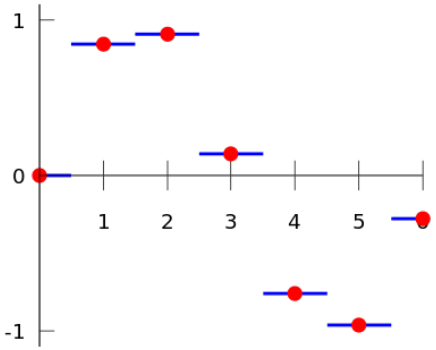
- no need to specify types in Python (“dynamically typed”)
- we can provide *type annotation* to hint at the expected type
- but Python interpreter does not check/enforce the type

Why (gradual) type annotations?

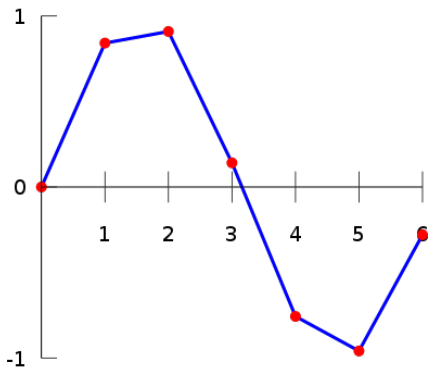
- contributes to documentation
- external tools can check typing (such as `mypy`)
- editors may use the information (e.g. for autocompletion)

Interpolation

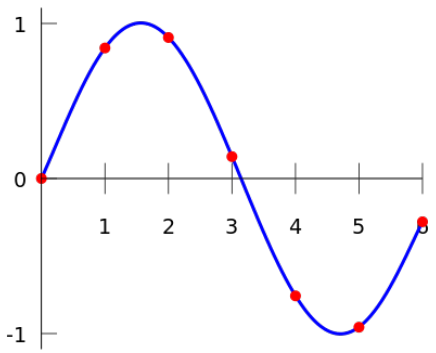
Interpolation of discrete data points



Piecewise constant interpolation



Linear interpolation



Polynomial interpolation

Source: https://en.wikipedia.org/wiki/Interpolation

Interpolation of data

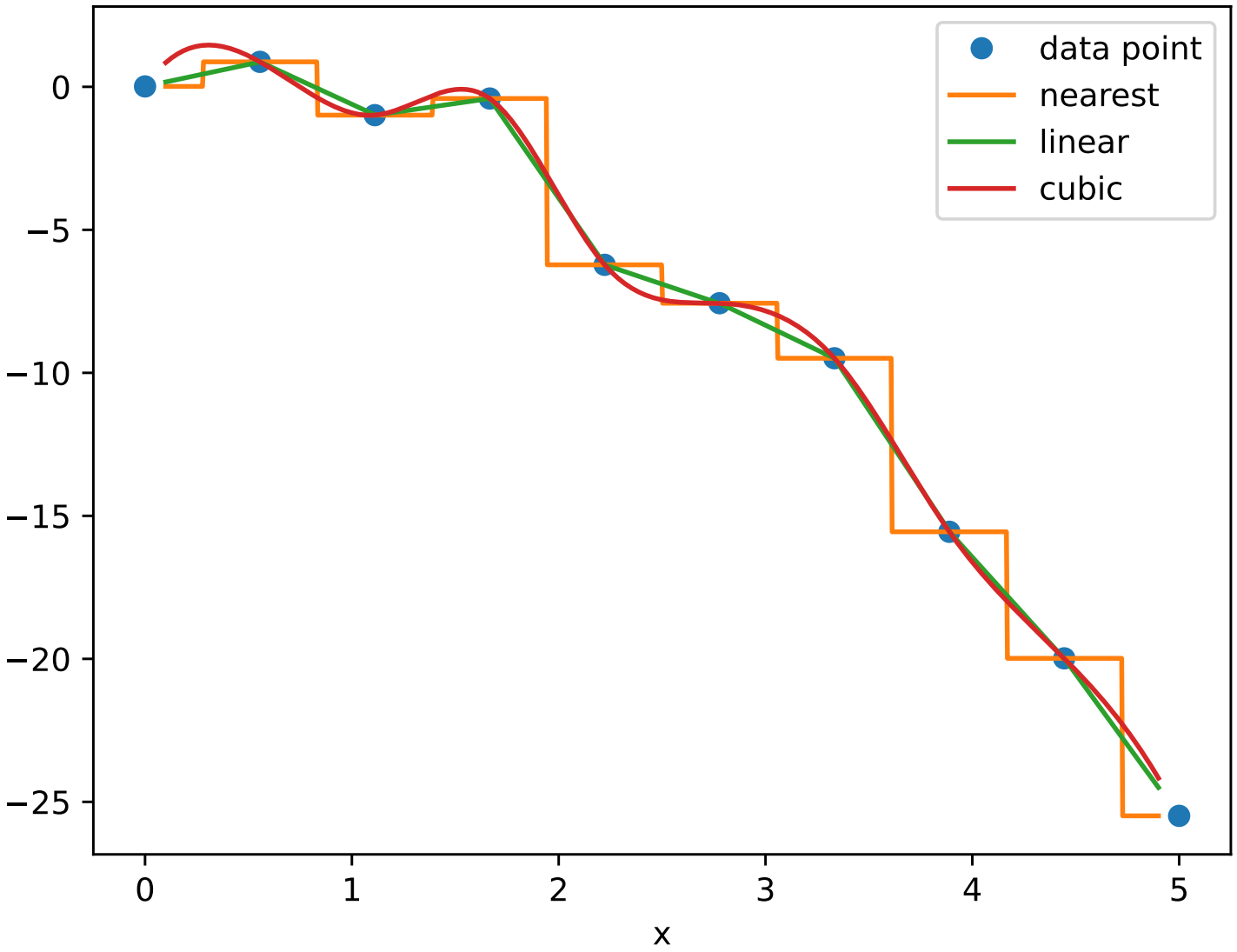
Given a set of N points (x_i, y_i) with $i = 1, 2, \ldots N$, we sometimes need a function $f(x)$ which returns $y_i = f(x_i)$ and interpolates the data between the x_i.

- $\rightarrow$ `y0 = scipy.interpolate.interp1d(x, y)` provides this interpolation
- `interp1d` returns *a callable* `y0` which interpolates the x-y data for any given x when called as `y0(x)`.
- Data interpolation of $y_i = f(x_i)$ may be useful to
  - create smoother plots of $f(x)$
  - find minima/maxima of $f(x)$
  - find x_c so that $f(x_c) = y_c$, provide inverse function $x = f^{-1}(y)$
  - integrate $f(x)$

Interpolation example

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.interpolate

def create_data(n):
    """Given an integer n, returns n data points x and values y as a numpy.array."""
    xmax = 5.0
    x = np.linspace(0, xmax, n)
    y = -x**2
    y += 1.5 * np.random.normal(size=len(x))  # make y-data somewhat irregular
    return x, y

n = 10
x, y = create_data(n)  # input data (such as a measurement)
xfine = np.linspace(0.1, 4.9, n * 100)  # use finer and regular mesh for plot
# interpolate with piecewise constant function (p=0):
y0 = scipy.interpolate.interp1d(x, y, kind="nearest")
# interpolate with piecewise linear func (p=1):
y1 = scipy.interpolate.interp1d(x, y, kind="linear")
# interpolate with cubic spline:
y2 = scipy.interpolate.interp1d(x, y, kind="cubic")

fig, ax = plt.subplots()
ax.plot(x, y, "o", label="data point")
ax.plot(xfine, y0(xfine), label="nearest")
ax.plot(xfine, y1(xfine), label="linear")
ax.plot(xfine, y2(xfine), label="cubic")
ax.legend(); ax.set_xlabel("x"); fig.savefig("interpolate.pdf")
```

Interpolation example



Closures

Returning function objects

We have seen that we can pass function objects as arguments to a function. Now we look at functions that *return function objects*.

Example (`closure_adder42.py`):

```
def make_add42():
    def add42(x):
        return x + 42
    return add42

add42 = make_add42()
print(add42(2))          # output is '44'
```

Closures

A closure (Wikipedia) is a function with bound variables. We often create closures by calling a function that returns a (specialised) function. For example (`closure_adder.py`):

```
import math

def make_adder(y):
    def adder(x):
        return x + y
    return adder

add42 = make_adder(42)
addpi = make_adder(math.pi)
print(add42(2))          # output is 44
print(addpi(-3))         # output is 0.14159265359
```

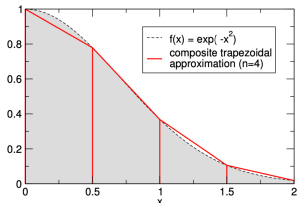
Common Computational Tasks

Overview working with functions and data

differentiation

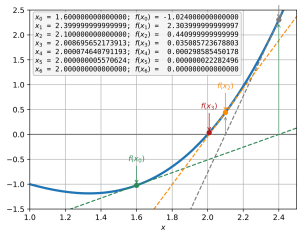
$$\frac{df}{dx}(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

integration



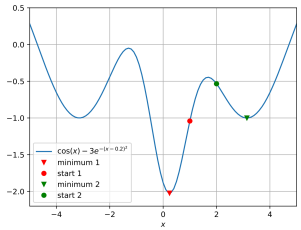
`quad`

root finding



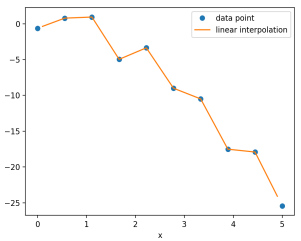
`brentq, solve`

optimisation



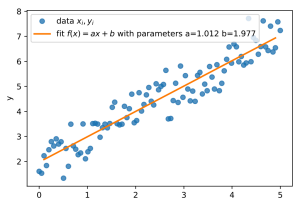
`fmin`

interpolation



`Interp1d`

curve fitting



`curve fitting`

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`)

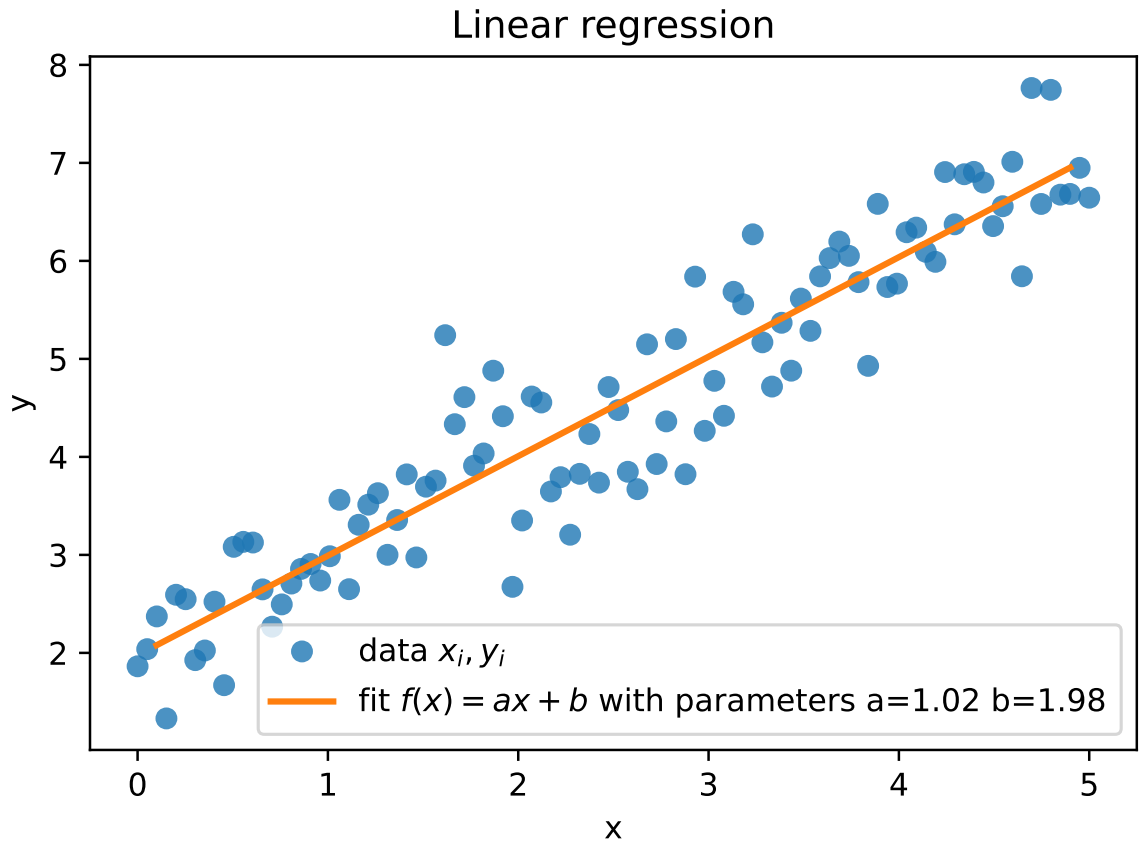
All in the following (third party) python packages:

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

Curve fitting

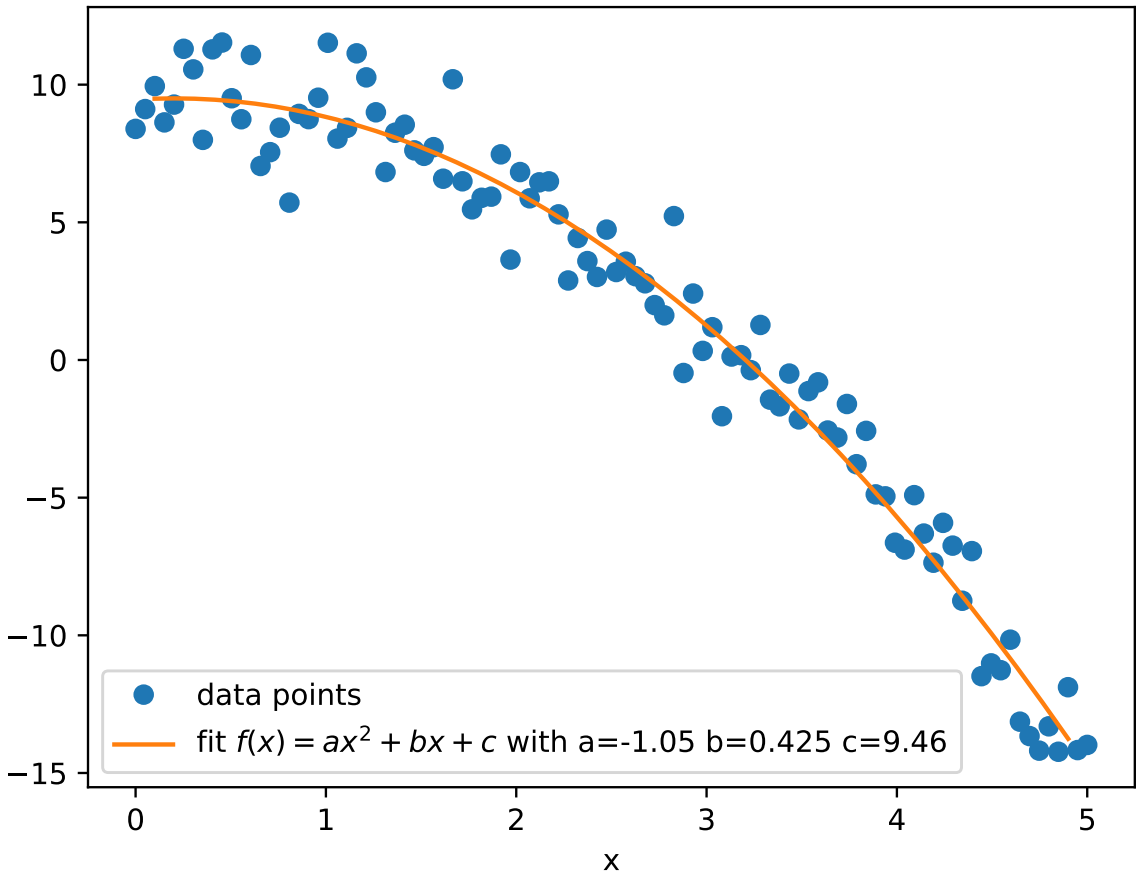
Curve fitting

Given n data points $(x_i, y_i), i = 1, \ldots, n$, and a model $y = f(x, \vec{p})$, with model parameters $\vec{p} = (p_1, p_2, \ldots)$, find coefficients $\vec{p}$ so that $y_i = f(x_i, \vec{p})$ describes the data "best".



Wikipedia: Curve fitting

Curve fitting example: parabola



Curve fitting example: parabola

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import scipy.optimize
4
5  def create_data(n):
6      """Given an integer n, returns n data points
7      x and values y as a numpy.array."""
8      xmax = 5.0
9      x = np.linspace(0, xmax, n)
10     y = -x**2 + 10  # i.e. a=-1 b=0 c=10
11     # make y-data somewhat irregular
12     y += 1.5 * np.random.normal(size=len(x))
13     return x, y
14
15 def model(x, a, b, c):  # Equation for fit
16     """Return ax^2 + bx +  c."""
17     return a * x ** 2 + b * x + c
18
```

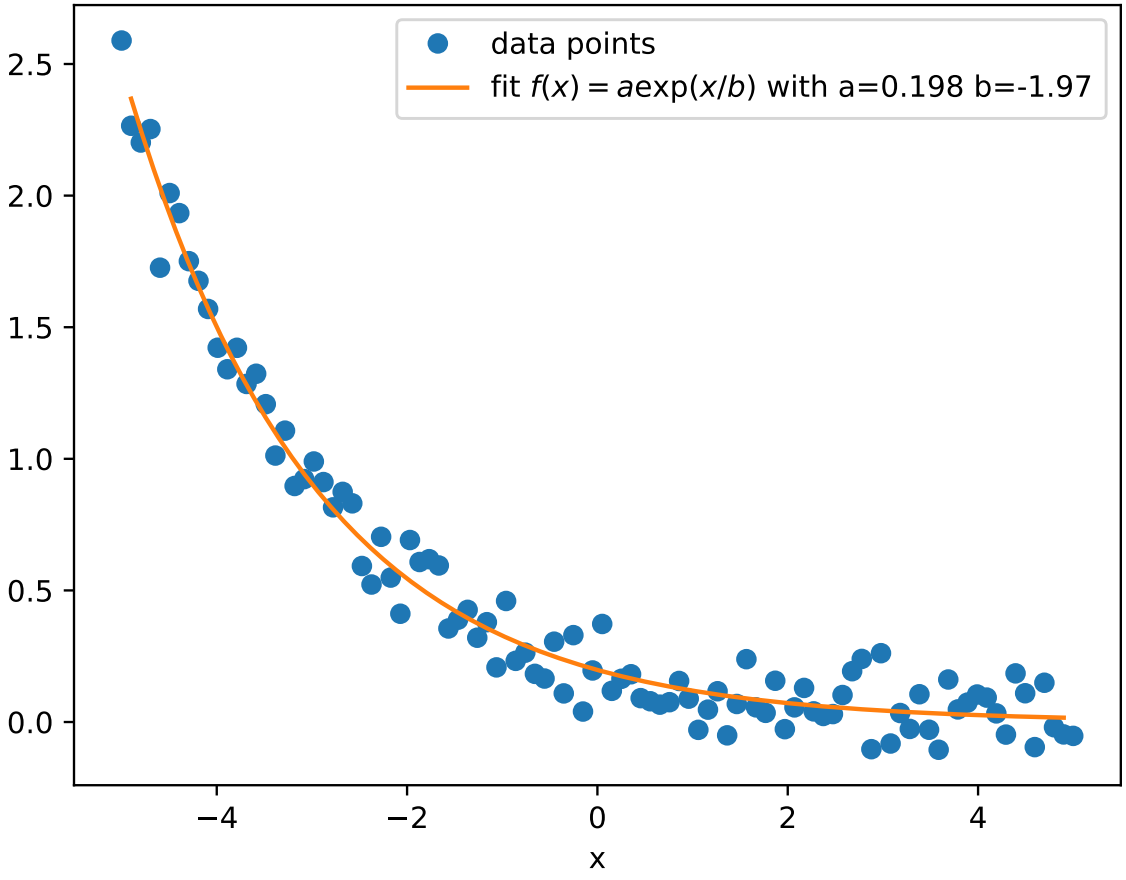


```
19 # main program
```

Curve fitting example: parabola

```
fig, ax = plt.subplots()
ax.plot(x, y, "o", label="data points")
label = fr"fit $f(x) = ax^2 + bx + c$ with {a=:.3} {b=:.3} {c=:.3}"
ax.plot(xfine, model(xfine, a, b, c), label=label)
ax.legend()
ax.set_xlabel("x")
fig.savefig("curvefit2.pdf")
```

Curve fitting example: exponential function



Curve fitting example: exponential function

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import scipy.optimize
4
5  def create_data(n):
6      """Given an integer n, returns n data points
7      x and values y as a numpy.array."""
8      xmax = 5.0
9      x = np.linspace(-xmax, xmax, n)
10     y = 0.2*np.exp(x/-2)  # i.e. a=0.2 b=-1
11     # make y-data somewhat irregular
12     y += 0.1 * np.random.normal(size=len(x))
13     return x, y
14
15 def model(x, a, b):  # Equation for fit
16     """Return a*exp(x/b)."""
17     return a * np.exp(x/b)
18
```

Curve fitting example: exponential function

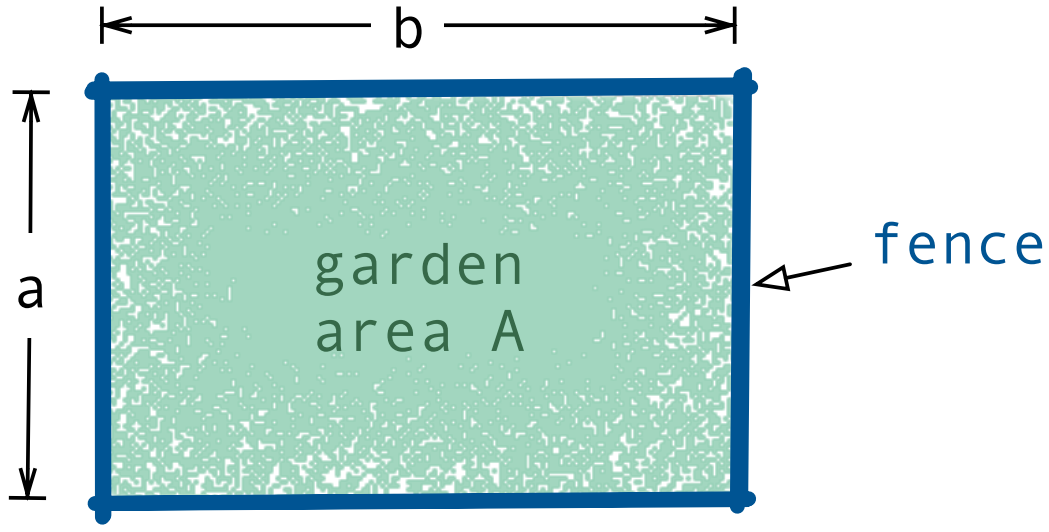
```
19  # main program
20  n = 100
21  x, y = create_data(n)
22  # do curve fit, and provide initial guess p0 = (a, b)
23  p, pcov = scipy.optimize.curve_fit(model, x, y, p0=(1, -1))
24  a, b = p  # extract result of curve_fit
25
26  # plot fit and data
27  xfine = np.linspace(-4.9, 4.9, n * 5)
28  fig, ax = plt.subplots()
29  ax.plot(x, y, "o", label="data points")
30  label = fr"fit $f(x) = a\exp(x/b)$ with {a=:.3} {b=:.3}"
31  ax.plot(xfine, model(xfine, a, b), label=label)
32  ax.legend()
33  ax.set_xlabel("x")
34  fig.savefig("curvefit3.pdf")
```

Curve fitting

- The curve fitting process is mapped onto a optimisation problem:
- Algorithm tries to minimise the error of the fit for the given data
- by varying the model parameters.
- A good initial guess (`p0`) may be needed.

Optimisation

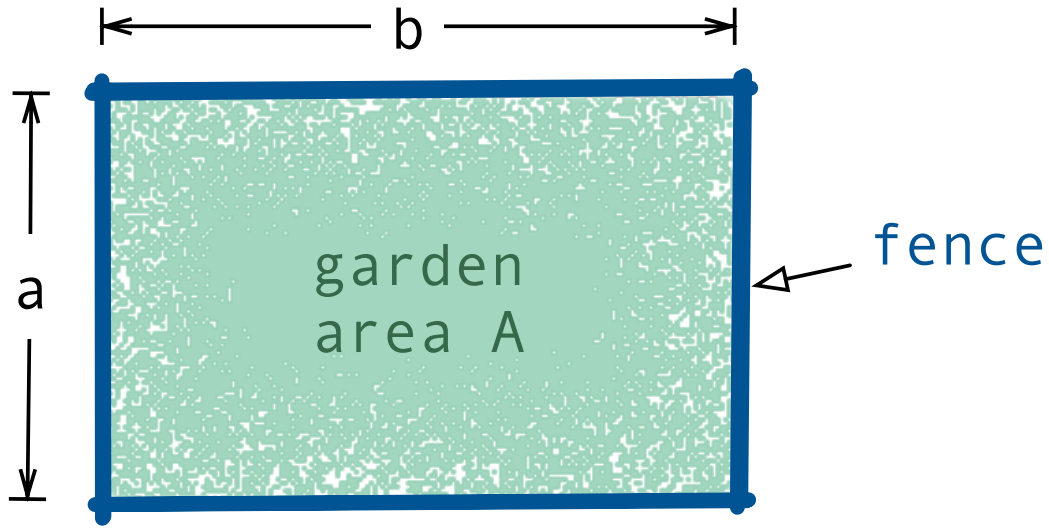
Optimisation example: garden fence



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



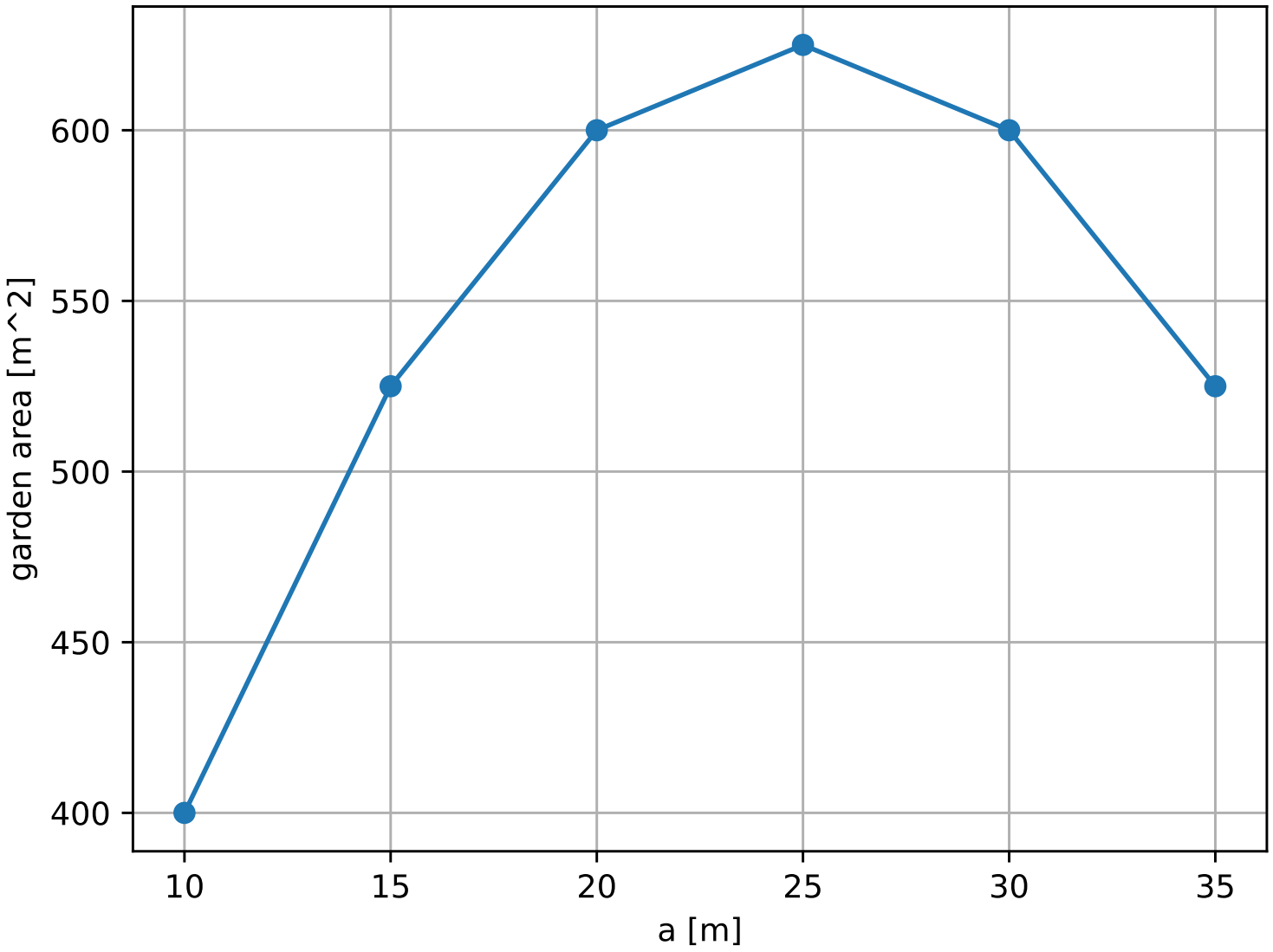
How do we find a and b that optimise the area $A(a, b)$?

- 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.

```
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
```

```
# 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 $\frac{\mathrm{d}A}{\mathrm{d}a} \stackrel{!}{=} 0 : \frac{\mathrm{d}A}{\mathrm{d}a} = \frac{L}{2} - 2a \Rightarrow a = \frac{L}{4}$
- $b = \frac{L-2a}{2} \Rightarrow b = \frac{L}{4}$
- Check $\frac{\mathrm{d}^2A}{\mathrm{d}a^2} = -2 < 0 \Rightarrow A\left(\frac{L}{4}\right)$ is maximum. ✓

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 $g(x)$.

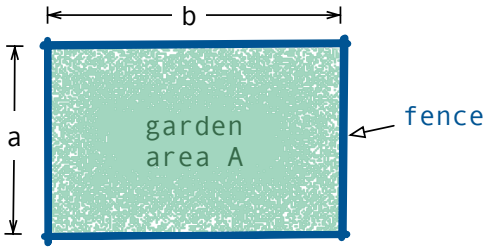
Optimisation example: parabola

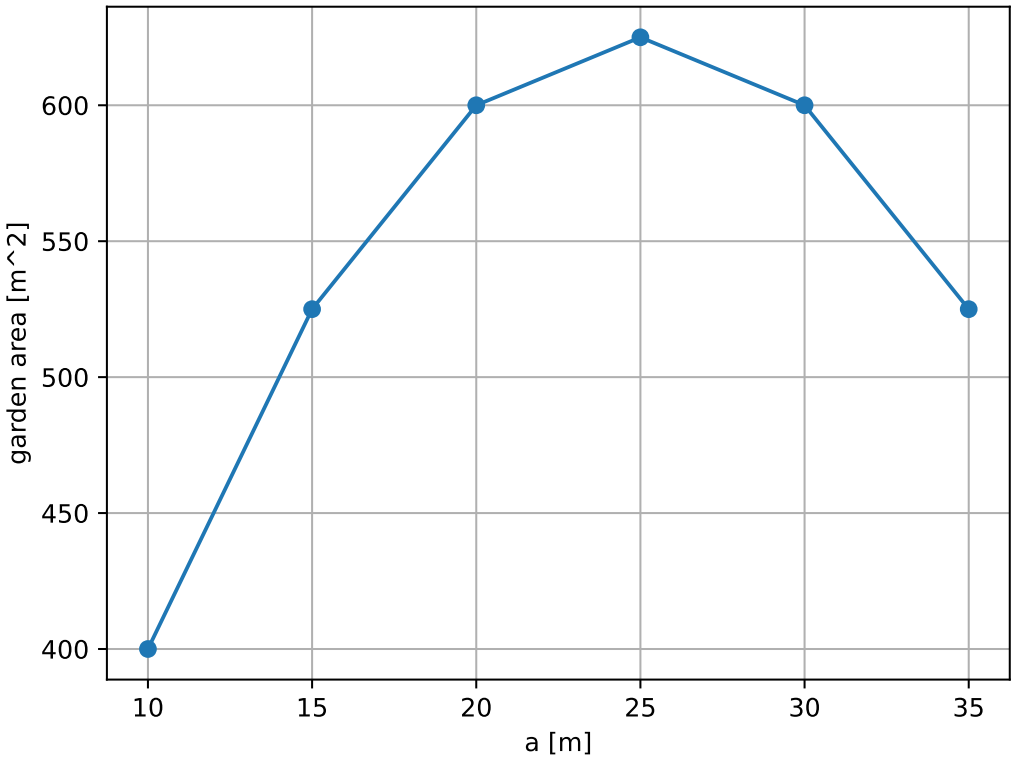
```
from scipy import optimize

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

```
1  import numpy as np
2  from scipy.optimize import fmin
3  import  matplotlib.pyplot as plt
4
5  def f(x):  # objective function
6      return np.cos(x) - 3 * np.exp(-((x - 0.2) ** 2))
7
8  # find minima of f(x),
9  # starting from 1.0 and 2.0 respectively
10 minimum1 = fmin(f, 1.0)
11 print("Start search at x=1., minimum is", minimum1)
12 minimum2 = fmin(f, 2.0)
13 print("Start search at x=2., minimum is", minimum2)
14
15 # plot function
16 x = np.arange(-10, 10, 0.1)
17 y = f(x)
18 fig, ax = plt.subplots()
19 ax.plot(x, y, label=r"$\cos(x)-3e^{-(x-0.2)^2}$")
20 ax.set_xlabel("$x$")
21 ax.set_xlabel("$f(x)$")
22 ax.grid()
23 ax.axis([-5, 5, -2.2, 0.5])
24
25 # add minimum1 to plot
```

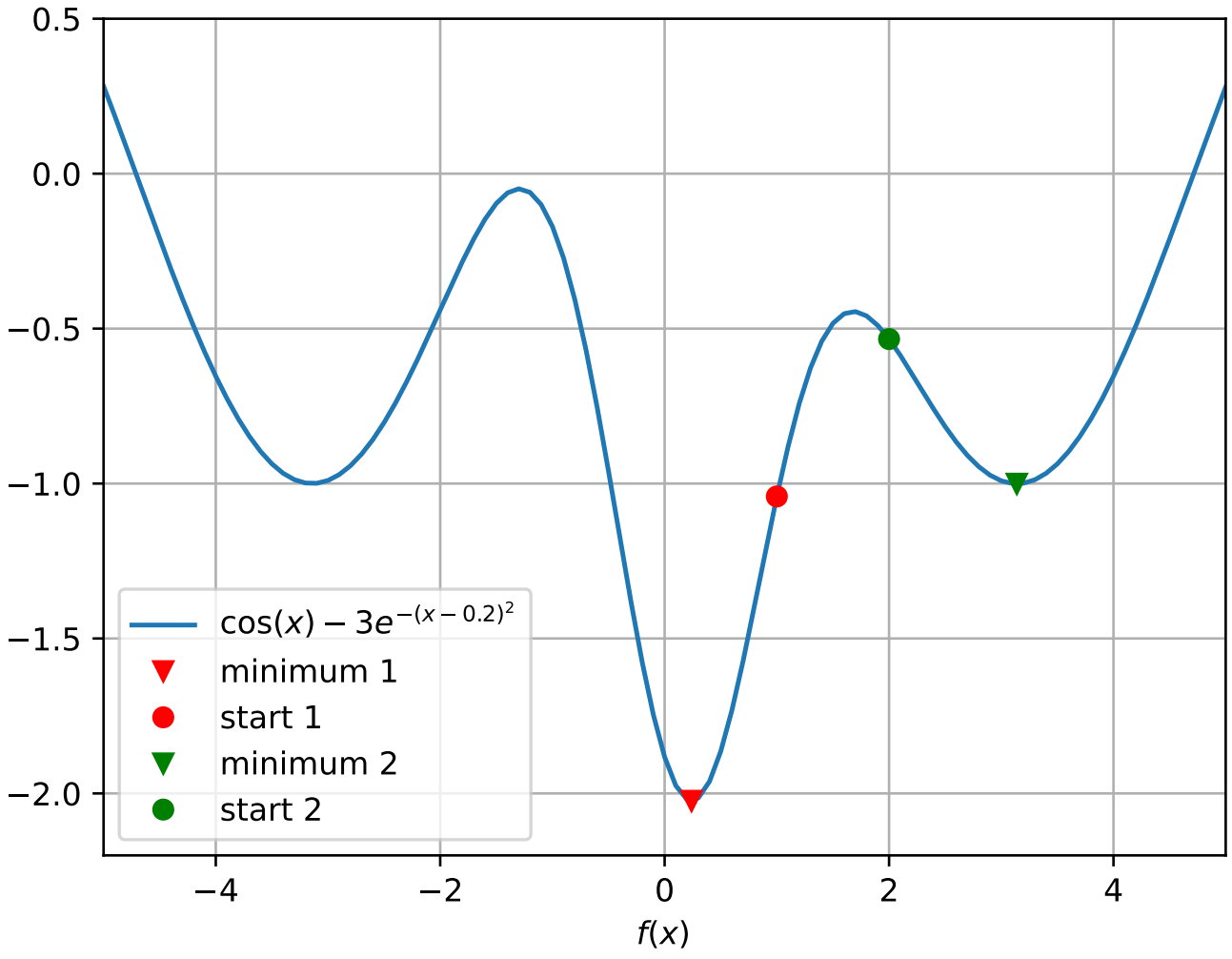
Optimisation example: multiple minima

```
26  ax.plot(minimum1, f(minimum1), "vr", label="minimum 1")
27  # add start1 to plot
28  ax.plot(1.0, f(1.0), "or", label="start 1")
29
30  # add minimum2 to plot
31  ax.plot(minimum2, f(minimum2), "vg", label="minimum 2")
32  # add start2 to plot
33  ax.plot(2.0, f(2.0), "og", label="start 2")
34
35  ax.legend(loc="lower left")
36  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



FIFO example and Object Oriented Programming (OOP)

Object Orientation (OO) and Closures

Earlier, we did an exercise for a first-in-first-out queue. At the time, we used a global variable to keep the state of the queue. To compare different approaches, the following slides show:

1. the original FIFO-queue solution (using a global variable, generally not good)
2. a modified version where the queue variable is passed to every function ($\rightarrow$ this is object oriented programming without objects)
3. an object oriented version (where the queue data is part of the queue object). Probably the best solution, see OO programming for details.
4. a version based on closures (where the state is part of the closures)

Original FIFO solution (`fifoqueue.py`)

```
queue = []
def length():
    """Returns number of waiting customers"""
    return len(queue)

def show():
    """print list of customers, longest waiting customer at end."""
    for name in queue:
        print(f"waiting customer: {name}")

def add(name):
    """Customer with name 'name' joining the queue"""
    queue.insert(0, name)

def next_():
    """Returns name of next to serve, removes customer from queue"""
    return queue.pop()

add('Spearing'); add('Fangohr'); add('Takeda')
show(); next_()
```

Improved FIFO solution (local variables)

Improved FIFO solution (`fifoqueue2.py`)

```
def length(queue):
    return len(queue)

def show(queue):
    for name in queue:
        print(f"waiting customer: {name}")

def add(queue, name):
    queue.insert(0, name)

def next_(queue):
    return queue.pop()

q1 = []
q2 = []
add(q1, 'Spearing');  add(q1, 'Fangohr');  add(q1, 'Takeda')
add(q2, 'John');  add(q2, 'Peter')
print(f"{length(q1)} customers in queue1:");  show(q1)
print(f"{length(q2)} customers in queue2:");  show(q2)
```

Object-Oriented FIFO solution (`fifoqueueOO.py`)

```
class Fifoqueue:
    def __init__(self):
        self.queue = []

    def length(self):
        return len(self.queue)

    def show(self):
        for name in self.queue:
            print(f"waiting customer: {name}")

    def add(self, name):
        self.queue.insert(0, name)

    def next_(self):
        return self.queue.pop()

q1 = Fifoqueue(); q2 = Fifoqueue()
q1.add('Spearing'); q1.add('Fangohr'); q1.add('Takeda')
q2.add('John'); q2.add('Peter')
print(f"{q1.length()} customers in queue1:"); q1.show()
```

*Functional (closure) FIFO solution (`fifoqueue_closure.py`)

```
def make_queue():
    queue = []
    def length():
        return len(queue)

    def show():
        for name in queue: print(f"waiting customer: {name}")

    def add(name):
        queue.insert(0, name)

    def next_():
        return queue.pop()
    return add, next_, show, length

q1_add, q1_next, q1_show, q1_length = make_queue()
q2_add, q2_next, q2_show, q2_length = make_queue()
q1_add('Spearing');  q1_add('Fangohr');  q1_add('Takeda')
q2_add('John');  q2_add('Peter')
print(f"{q1_length()} customers in queue1:"); q1_show()
print(f"{q2_length()} customers in queue2:"); q2_show()
```

*Advanced: Using double-ended-queue (deque)

Specialised *double-ended-queue* data structure `deque` [1] available in the `collections` module of python:

[1] https://docs.python.org/3/library/collections.html#collections.deque

```
from collections import deque

def length(queue):
    return len(queue)

def show(queue):
    for name in queue:
        print(f"waiting customer: {name}")

def add(queue, name):
    queue.appendleft(name)

def next_(queue):
    return queue.pop()

q1 = deque()
add(q1, 'Spearing');  add(q1, 'Fangohr');  add(q1, 'Takeda')
```

Lessons (Object Orientation)

Object orientation (OO):

- one important idea is to combine data and functions operating on data (in objects),
- objects contain data but
- access to data through interface (implementation details irrelevant to user)
- can program in OO style without OO-programming language:
  - as in FIFO2 solution
  - as in closure based approach
- OO mainstream programming paradigm (Java, C++, C#, ...)
- Python supports OO programming, and all things in Python are objects (see also slides 355 pp)

Functional tools: lambda, map, filter, reduce

More list processing and functional programming

- So far, have processed lists by iterating through them using for-loop
- perceived to be conceptually simple (by most learners) but
- not as compact as possible and not always as fast as possible
- Alternatives:
  - list comprehension
  - `map`, `filter`, `reduce`, often used with `lambda`

Anonymous function lambda

- lambda: anonymous function (function literal)
- Useful to define a small helper function that is only needed once

Anonymous function lambda

```
>>> lambda a: a
<function <lambda> at 0x319c70>
>>> lambda a: 2 * a
<function <lambda> at 0x319af0>
>>> (lambda a: 2 * a)
<function <lambda> at 0x319c70>
>>> (lambda a: 2 * a)(10)
20
>>> (lambda a: 2 * a)(20)
40
>>> (lambda x, y: x + y)(10, 20)
30
>>> (lambda x, y, z: (x + y) * z )(10, 20, 2)
60
>>> type(lambda x, y: x + y)
<type 'function'>
```

Lambda usage example 1

Integrate $f(x) = x^2$ from 0 to 2 (numerically):

- Without lambda (`lambda1.py`):

```
from scipy.integrate import quad
def f(x):
    return x**2

y, abserr = quad(f, a=0, b=2)
print(f"value is {y:f} +- {abserr:g}")
```

- With lambda (`lambda1b.py`):

```
from scipy.integrate import quad
y, abserr = quad(lambda x: x**2, a=0, b=2)
print(f"value is {y:f} +- {abserr:g}")
```

Output (same for both programs):

```
value is 2.666667 +- 2.96059e-14
```

Higher order functions

Roughly: “Functions that take or return functions” (see for example Wikipedia entry)

Rough summary (check `help(COMMAND)` for details)

- `map(function, iterable)` → `iterable`:
  apply function to all elements in iterable
- `filter(function, iterable)` → `iterable`:
  return items of iterable for which `function(item)` is true.
- `reduce(function, iterable, initial)` → `value`:
  apply function(x,y) from left to right to reduce iterable to a single value.

Note that sequences are iterables.

Map

- `map(function, iterable)` → `iterable`:
  apply `function` to all elements in sequence
- Example:

```
>>> def f(x):
...     return x ** 2
>>> map(f, [0, 1, 2, 3, 4])
<map object at 0x1026a52e8>             # this is iterable
>>> list(map(f, [0, 1, 2, 3, 4]))      # convert to list
[0, 1, 4, 9, 16]
```

- `lambda` converts an expression (`x ** 2`) to a function:

```
>>> list(map(lambda x: x ** 2, [0, 1, 2, 3, 4]))
[0, 1, 4, 9, 16]
```

- Equivalent operation using list comprehension:

```
>>> [x ** 2 for x in [0, 1, 2, 3, 4]]
[0, 1, 4, 9, 16]
```

Examples map

- Example (maths):

```
>>> import math
>>> list(map(math.exp, [0, 0.1, 1.]))
[1.0, 1.1051709180756477, 2.718281828459045]
```

- Example (slug):

```
>>> news="Python programming occasionally \
... more fun than expected"
>>> slug = "-".join(map(
... lambda w: w[0:6], news.split()))
>>> slug
'Python-progra-occasi-more-fun-than-expect'
```

Equivalent list comprehension expression:

```
>>> slug = "-".join([w[0:6] for w in news.split()])
```

Filter

`filter(function, iterable)` → `iterable`:

return items of iterable for which function(item) is true:

```
>>> def is_positive(n):  # returns True for positive n
...     return n > 0
>>> list(filter(is_positive,
...             [-3, -2, -1, 0, 1, 2, 3, 4]))
[1, 2, 3, 4]
>>> list(filter(lambda n: n > 0,
...             [-3, -2, -1, 0, 1, 2, 3, 4]))
[1, 2, 3, 4]
```

List comprehension equivalent:

```
>>> [n for n in [-3, -2, -1, 0, 1, 2, 3, 4] if n > 0]
[1, 2, 3, 4]
```

Examples filter

```
>>> c = "The quick brown fox jumps".split()
>>> print(c)
['The', 'quick', 'brown', 'fox', 'jumps']
>>> def len_gr_4(s):  # return True if s has >4 letters
...     return len(s) > 4
>>> list(map(len_gr_4, c))
[False, True, True, False, True]
>>> filter(len_gr_4, c)
<filter object at 0x10522e5c0>
>>> list(filter(len_gr_4, c))
['quick', 'brown', 'jumps']
>>> list(filter(lambda s: len(s) > 4, c)
['quick', 'brown', 'jumps']
```

Examples filter

Equivalent operation using list comprehension:

```
>>> [s for s in c if len(s) > 4]
['quick', 'brown', 'jumps']
```

Reduce

- `functools.reduce(function, iterable, initial)` → `value`:
  apply function(x, y) from left to right to reduce iterable to a single value.
- Examples:

```
>>> from functools import reduce
>>> def f(x, y):
...     print(f"Called with {x=}, {y=}")
...     return x + y
...
>>> reduce(f, [1, 3, 5], 0)
Called with x=0, y=1
Called with x=1, y=3
Called with x=4, y=5
9
```

Reduce

```
>>> reduce(f, [1, 3, 5], 100)
Called with x=100, y=1
Called with x=101, y=3
Called with x=104, y=5
109
>>> reduce(f,"test","")
Called with x=, y=t
Called with x=t, y=e
Called with x=te, y=s
Called with x=tes, y=t
'test'
>>> reduce(f,"test","FIRST")
Called with x=FIRST, y=t
Called with x=FIRSTt, y=e
Called with x=FIRSTte, y=s
Called with x=FIRSTtes, y=t
'FIRSTtest'
```

*Operator module

- operator module contains functions which are typically accessed not by name, but via some symbols or special syntax.
- For example `3 + 4` is equivalent to `operator.add(3, 4)`. Thus:

```
def f(x, y):
    return x + y
reduce(f, range(10), 0)
```

can also be written as:

*Operator module

```
reduce(operator.add, range(10), 0)
```

Note: could also use:

```
reduce(lambda x, y: x + y, range(10), 0)
```

but use of `operator` module is preferred (often faster) as the functionality is already provided in a function.

Functional programming

- Functions like `map`, `reduce` and `filter` are found in just about any lanugage supporting functional programming.
- provide functional abstraction for commonly written loops
- Use those (and/or list comprehension) instead of writing loops, because
  - Writing loops by hand is quite tedious and error-prone.
  - The functional version is often clearer to read.
  - The functional version can result in faster code (if you can avoid `lambda`)

What command to use when?

- `lambda` allows to define a (usually simple) function "in-place". We need this to convert an *expression* into a *function*.
- `map` transforms a sequence to another sequence (of same length) using a function
- `filter` filters a sequence (reduces number of elements) using a function
- `list comprehension` transforms a list (can include filtering) using an expression
  - if you need to use a `lambda` in a `map`, you are probably better off using list comprehension.
  - if you have a function to apply, `map` is more compact than a list comprehension.
- `reduce` carries out an operation that "collects" information (sum, product, ...), for example reducing the sequence to a single number.

Example: squaring elements in list with expression `x**2`

Some alternatives:

```
>>> res = []
>>> for x in range(5):
...     res.append(x ** 2)
...
>>> res
[0, 1, 4, 9, 16]

>>> [x ** 2 for x in range(5)]
[0, 1, 4, 9, 16]

>>> list(map(lambda x: x ** 2, range(5)))
[0, 1, 4, 9, 16]
```

Example: squaring elements in list with function `f`

```
>>> def f(x):
...     return x**2

>>> res = []
>>> for x in range(5):
...     res.append(f(x))
...
>>> res
[0, 1, 4, 9, 16]

>>> [f(x) for x in range(5)]
[0, 1, 4, 9, 16]

>>> list(map(f, range(5)))
[0, 1, 4, 9, 16]
```

Scientific Python

SciPy (SCIentific PYthon)

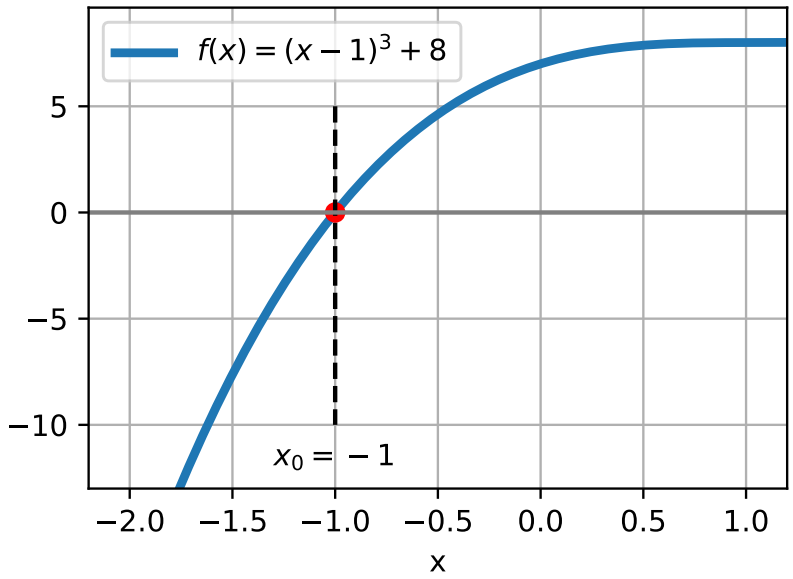
(Partial) output of `help(scipy)`:

```
constants     --- Physical and math. constants and units
integrate     --- Integration routines
interpolate   --- Interpolation Tools
io            --- Data input and output (also matlab)
linalg        --- Linear algebra routines
ndimage       --- N-D image package
optimize      --- Optimization Tools
signal        --- Signal Processing Tools
sparse        --- Sparse Matrices
spatial       --- Spatial data structures and algorithms
special       --- Special functions
stats         --- Statistical Functions
```

Root finding

Root finding

- Given a function $f(x)$,
- we are searching an x_0 so $f(x_0) = 0$.
- We call x_0 a root of $f(x)$.

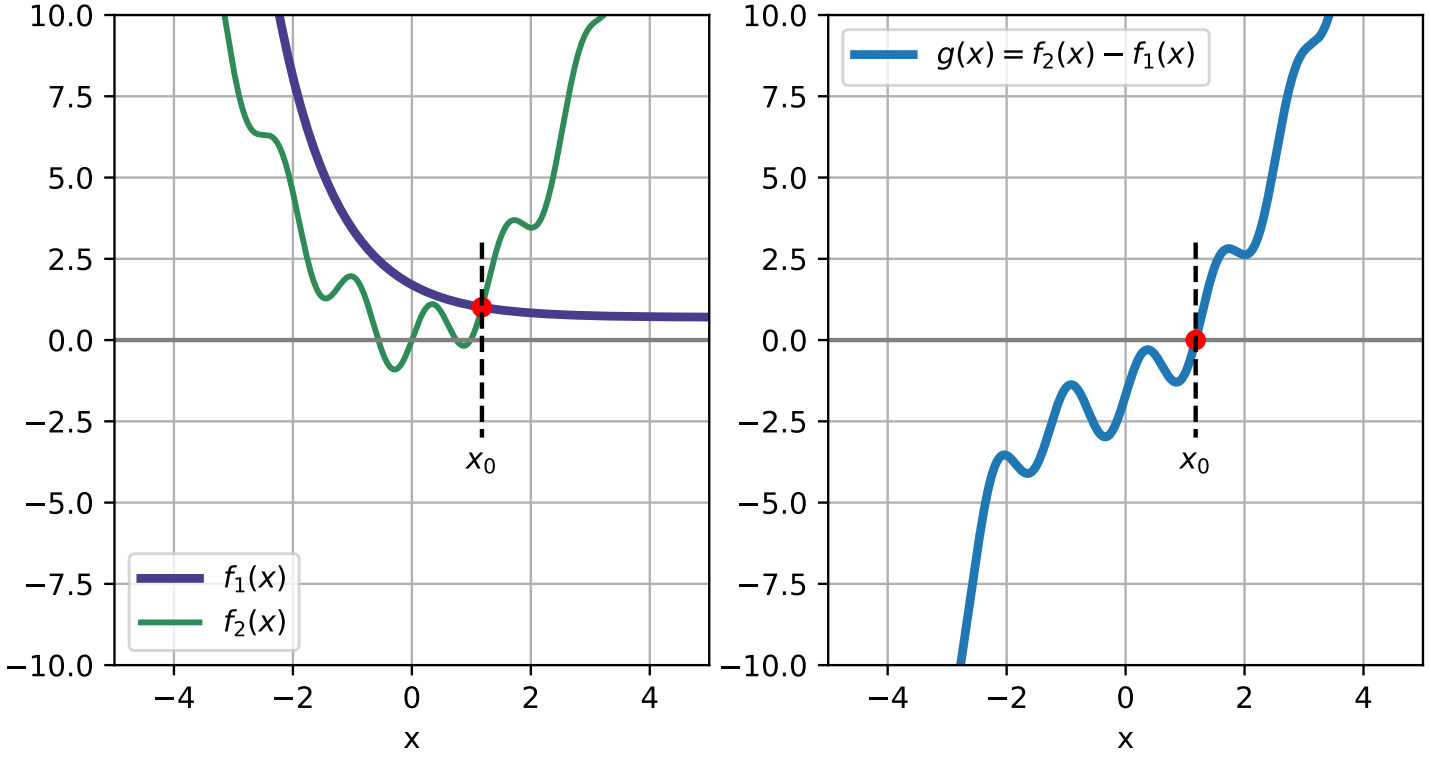


Why?

- Many science and engineering problems lead to equations of the type $f(x) = 0$

Rootfinding: find crossing of two functions

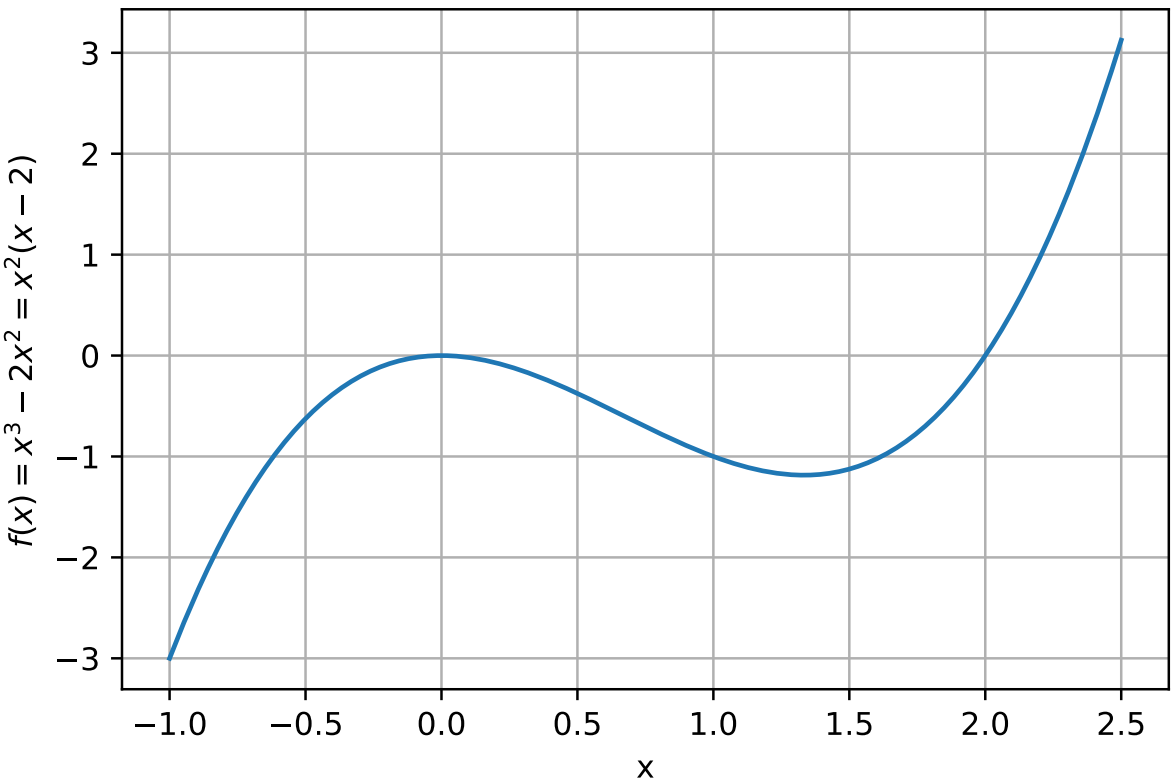
- Often we have two functions $f_1(x)$ and $f_2(x)$, and we are looking for x_0 so that $f_1(x_0) = f_2(x_0)$ (red dot, left plot).
- in that case, we define $g(x) = f_2(x) - f_1(x)$ and find a root for $g(x)$ (red dot, right plot)



Note that $f_2(x)$ could be a constant, such as $f_2(x) = 100$ if we want to find the value x_0 for which $f_1(x_0) = 100$.

Example

- Find root of function $f(x) = x^2(x-2)$
- f has a double root at $x = 0$, and a single root at $x = 2$.
- Ask algorithm to find single root at $x = 2$.



Using BrentQ algorithm from `scipy`

```
from scipy.optimize import brentq

def f(x):
    """returns f(x)=x^3-2x^2. Has roots at
    x=0 (double root) and x=2"""
    return x ** 3 - 2 * x ** 2

# main program starts here
x = brentq(f, a=1.5, b=3, xtol=1e-6)

print(f"Root is approx {x}.")
print(f"The exact error is {2-x}.")
```

produces:

```
Root is approx 2.0000000189582865.
The exact error is -1.895828649622899e-08.
```

Rootfinding for $f(x) = 0$ (scalar x): BrentQ

- To solve $f(x) = 0$ with o scalar x, we recommend the BrentQ method
- Assumptions:
  - We have the function f available as a Python function
  - The function f has a single root between a and b
  - The function is continuous
- The BrentQ method
  - will find and return the root $x \in [a, b]$
  - will use a fast (Newton) method if possible.

Root finding summary

- Given the function f(x), applications for root finding include:
  - to find x_1 so that $f(x_1) = y$ for a given y (this is equivalent to computing the inverse of the function f).
  - to find crossing point x_c of two functions $f_1(x)$ and $f_2(x)$ (by finding root of difference function $g(x) = f_1(x) - f_2(x)$)
- Recommended method: `scipy.optimize.brentq` which combines the safe feature of the bisect method with the speed of the Newton method.
- *For multi-dimensional functions $f(\mathbf{x})$, use `scipy.optimize.fsolve`.

*Using `fsolve` for multi-dimensional root-finding problem

```
from scipy.optimize import fsolve  # multidimensional solver

def f(v):
    """Return f(x, y) = (x^3, y). Trivial example with
    root at x=0 and y=-1"""
    x, y = v
    return x**3, y+1

x, y = fsolve(f, x0=[2, 2])  # start search from x=2, y=2
print(f"Root is approximately at\nx={float(x)} "
      f"and y={float(y)}")
```

produces:

```
Root is approximately at
x=1.0586069199901217e-16 and y=-1.0
```

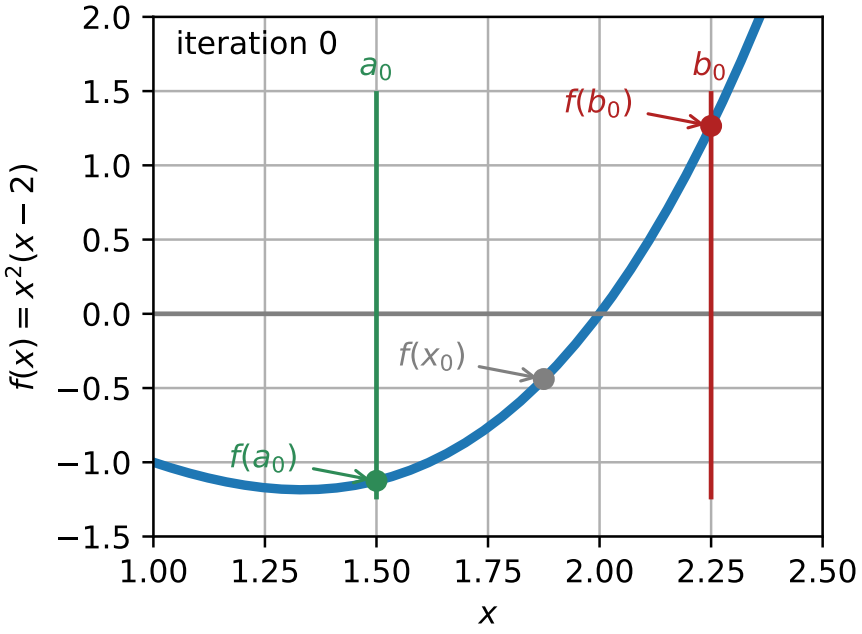
The bisection algorithm

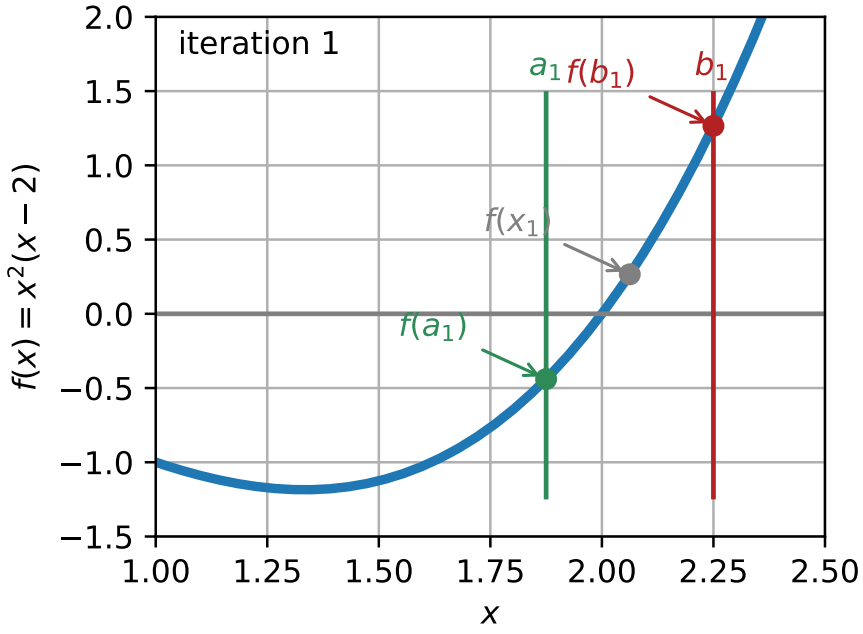
- Function: `bisect(f, a, b)`
- Assumptions:
  - Given: a (float)
  - Given: b (float)
  - Given: $f(x)$, continuous with single root in $[a, b]$, i.e. $f(a)f(b) < 0$
  - Given: f_{tol} (float), for example $f_{\text{tol}} = 10^{-6}$

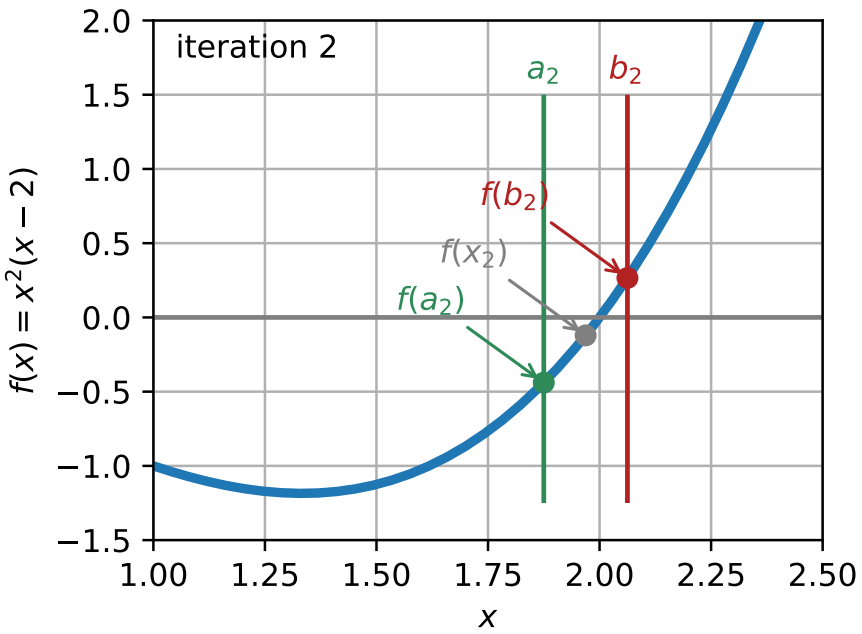
The bisection method returns x so that $|f(x)| < f_{\text{tol}}$.

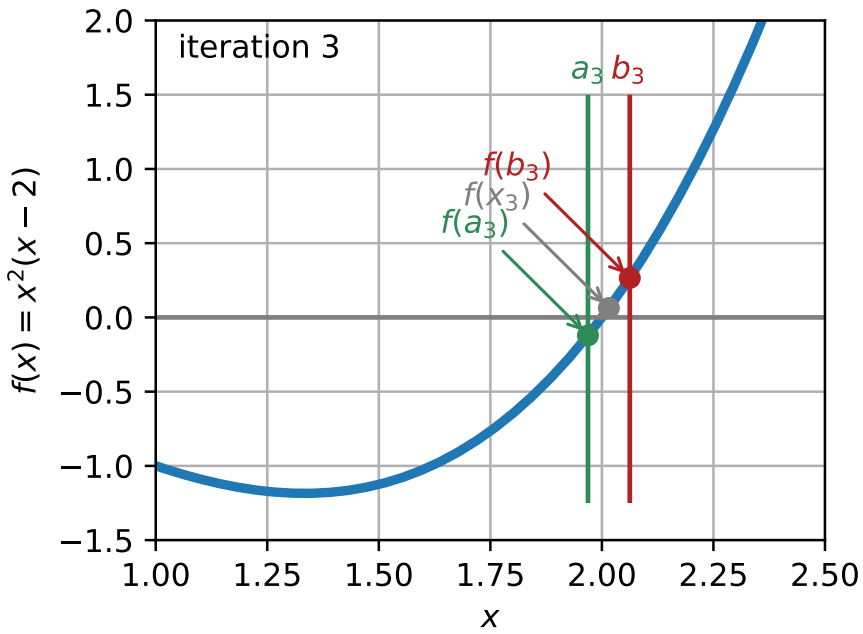
1. $x = (a + b)/2$
2. while $|f(x)| > f_{\text{tol}}$ do
   - if $f(x)f(a) > 0$
     then $a \leftarrow x$ # throw away left half
     else $b \leftarrow x$ # throw away right half
   - $x = (a + b)/2$
3. return x

The bisection algorithm









The bisection function from `scipy`

- Scientific Python provides an interface to the “Minpack” library. One of the functions is
- `scipy.optimize.bisect(f, a, b[, xtol])`
  - `f` is the function for which we search x such that $f(x) = 0$
  - `a` is the lower limit of the bracket $[a, b]$ around the root
  - `b` is the upper limit of the bracket $[a, b]$ around the root
  - `xtol` is an *optional* parameter that can be used to modify the default accuracy of $x_{\mathrm{tol}} = 10^{-12}$
- the `bisect` function stops ’bisecting’ the interval around the root when $|\texttt{b} - \texttt{a}| < \texttt{xtol}$.

Using bisection algorithm from `scipy`

```
from scipy.optimize import bisect

def f(x):
    """returns f(x)=x^3-2x^2. Has roots at
    x=0 (double root) and x=2"""
    return x ** 3 - 2 * x ** 2

# main program starts here
x = bisect(f, a=1.5, b=3, xtol=1e-6)

print(f"Root is approx {x}.")
print(f"The exact error is {x-2}.")
print(f"Error is less than 1e-6: {abs(x-2)<1e-6}")
```

produces

```
Root is approx 2.000000238418579.
The exact error is 2.3841857910156250e-07.
Error is less than 1e-6: True
```

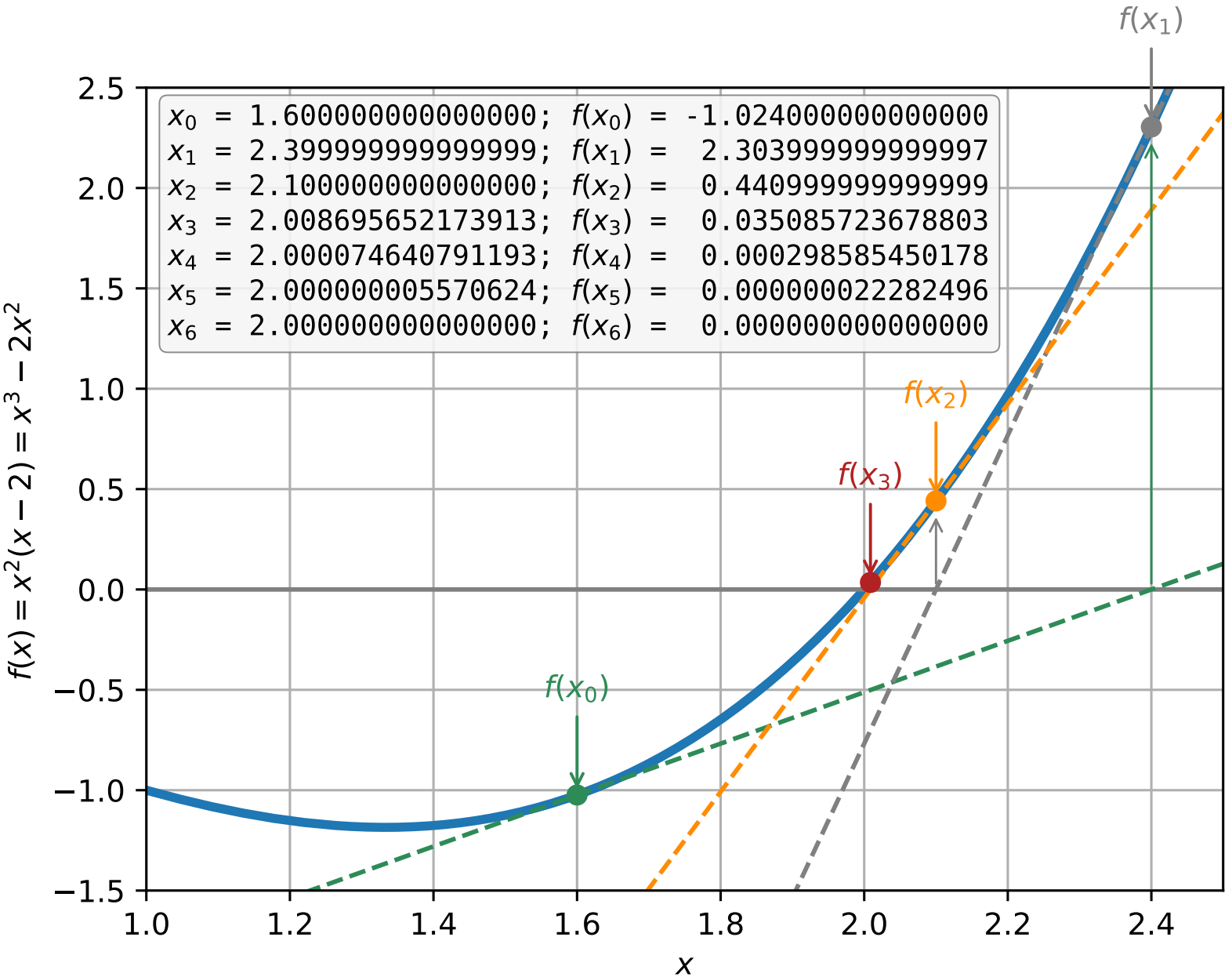
Rootfinding: the Newton method

- Aim: find x_{root} so that $f(x_{\mathrm{root}}) = 0$.
- Idea: close to the root x_{root}, the tangent of $f(x)$ is likely to point to the root. Make use of this information.
- Algorithm:
  while $|f(x)| >$ftol, do

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

  where $f'(x) = \frac{\mathrm{d}f}{\mathrm{d}x}(x)$.
- fast convergence (much better than bisection method)
- but not guaranteed to converge.
- Need a good initial guess x_0 for the root.
- Need a way to compute (or approximate) $f'(x) \equiv \frac{\mathrm{d}f}{\mathrm{d}x}(x)$.

The Newton method (`tol=1e-15`)



Comparison Bisection & Newton method

Bisection method

- Requires root in bracket $[a, b]$
- guaranteed to converge (for single roots)
- Library function: `scipy.optimize.bisect`

Newton method

- Requires good initial guess x for root x_0
- may never converge
- but if it does, it is quicker than the bisection method
- Library function: `scipy.optimize.newton`

In practice, start with `brentq`, which combines advantages of `bisect` and `newton`.

Computing derivatives numerically

Overview

Motivation:

- We need derivatives of functions for some optimisation and root finding algorithms
- Not always is the function analytically known (but we are usually able to compute the function numerically)
- The material presented here forms the basis of the finite-difference technique that is commonly used to solve ordinary and partial differential equations.

From analytical maths to numerics: 1st derivative

- One definition of derivative (or "*differential* operator" $\frac{\mathrm{d}}{\mathrm{d}x}$):

$$f'(x) = \frac{\mathrm{d}f}{\mathrm{d}x}(x) = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$

- Use *difference* operator to approximate *differential* operator

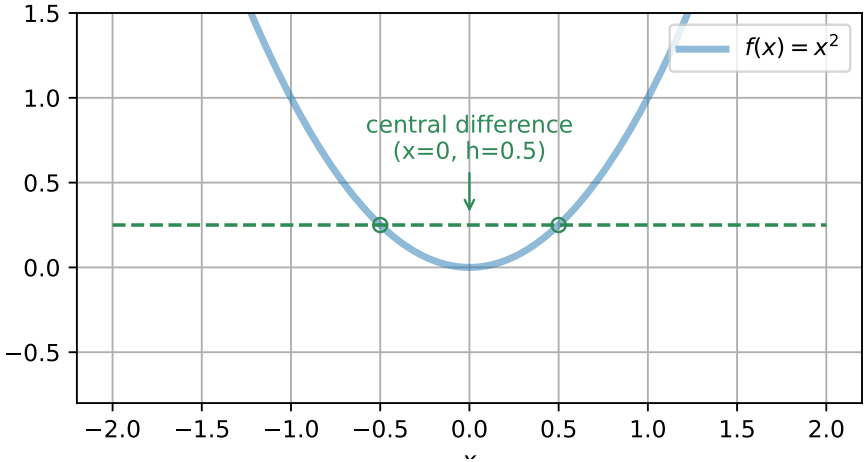
$$f'(x) = \frac{\mathrm{d}f}{\mathrm{d}x}(x) = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h} \approx \frac{f(x+h) - f(x-h)}{2h}$$

- $\Rightarrow$ can now compute *an approximation* of $f'(x)$ simply by evaluating $f(x+h)$ and $f(x-h)$.
- We can choose h. Make it small (perhaps 10^{-6}), but not too small (10^{-15}).

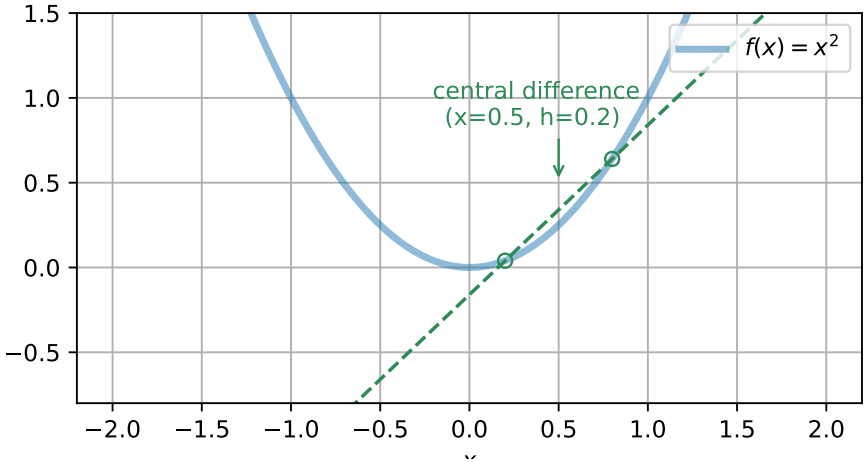
Geometric representations finite difference approximation

central difference approximation of derivative

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h}$$



$f'(0)$ with $h = 0.5$



$f'(0.5)$ with $h = 0.2$

Example $f(x) = \frac{1}{3}x^3$

- Derivative of $f(x) = x^3/3$:

$$f'(x) = \frac{\mathrm{d}}{\mathrm{d}x} f(x) = \frac{\mathrm{d}}{\mathrm{d}x} \frac{x^3}{3} = x^2$$

- Central differences approximation at $x = 2$ with $h = 0.1$:

$$\begin{aligned} f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} &= \frac{\frac{1}{3}(x+h)^3 - \frac{1}{3}f(x-h)^3}{2h} \\ &= \frac{1}{3}\frac{2.1^3 - 1.9^3}{2h} \\ &= \frac{1}{3}\frac{2.1^3 - 1.9^3}{0.2} = 4.0033333\ldots \end{aligned}$$

*spacing `h` in central differences

Compute central difference approximation of

$$\frac{\mathrm{d}}{\mathrm{d}x}\frac{x^3}{3} = x^2$$

at $x = 2$. Correct result is $x^2 = 2^2 = 4$.

Try different values of spacing h:

```
     h      centr. diff. appr    abs. error
-------------------------------------------
   0.1      4.003333333333337   0.00333333
 0.001      4.000000333332698   3.33333e-07
 1e-06      4.000000000115023   1.15023e-10
 1e-07      3.999999997894577   2.10542e-09
 1e-09      4.000000330961484   3.30961e-07
 1e-12      4.000355602329364   0.000355602
 1e-15      3.996802888650563   0.00319711
```

→ too large h: inaccurate approximation of derivative

→ too small h: floating point representation errors

*Example: spacing `h` in central differences

```
def f(x):
    """Return x^3/3. (Derivative is x^2)."""
    return x**3 / 3

x = 2
exact = 2**2  # # correct derivative of x^3/3 at x=2 is 4
print("       h     centr. diff. appr   abs. error")
print("   ---------------------------------------")
for h in [1e-1, 1e-3, 1e-6, 1e-7, 1e-9, 1e-12, 1e-15]:
    fprime = (f(x+h) - f(x-h)) / (2 * h)
    print(f"{h:8g}  {fprime:20.15f}  {abs(fprime-exact):10.6g}")
```

Summary

- Can approximate derivatives of $f(x)$ numerically
- need only function evaluations of $f(x)$
    - $f(x)$ could be measured or simulated data, for example.

Numerical Integration of (math) functions

Function integration example

Aim: Compute

$$I = \int_a^b f(x)\mathrm{d}x, \qquad \text{with} \quad a = -2, b = 2$$

and

$$f(x) = \exp(-\cos(2x\pi)) + 3.2$$

```
from math import cos, exp, pi
from scipy.integrate import quad

# function we want to integrate
def f(x):
    return exp(cos(-2 * x * pi)) + 3.2

# call quad to integrate f from -2 to 2
res, err = quad(f, -2, 2)

print(f"The numerical result is {res:f} (+-{err:g})")
```

```
The numerical result is 17.864264 (+-1.55117e-11)
```

Numerical Integration

Numerical Integration 1— Overview

Different situations where we use integration:

(A) solving (definite) integrals

(B) solving (ordinary) differential equations

- more complicated than (A)
- Euler's method, Runge-Kutta methods

Both (A) and (B) are important.

We begin with the numeric computation of integrals (A).

(A) Definite Integrals

Often written as

$$I = \int_a^b f(x)\mathrm{d}x \tag{7}$$

- example: $I = \int_0^2 \exp(-x^2)\mathrm{d}x$
- solution is $I \in \mathbb{R}$ (*i.e.* a number)
- right hand side $f(x)$ depends only on x
- if $f(x) > 0 \quad \forall x \in [a, b]$, then we can visualise I as the area underneath $f(x)$
  - Note that the integral is *not* necessarily the same as the area enclosed by $f(x)$ and the x-axis:
    - $\int_0^{2\pi} \sin(x)\mathrm{d}x = 0$
    - $\int_0^1 (-1)\mathrm{d}x = -1$

Often written as

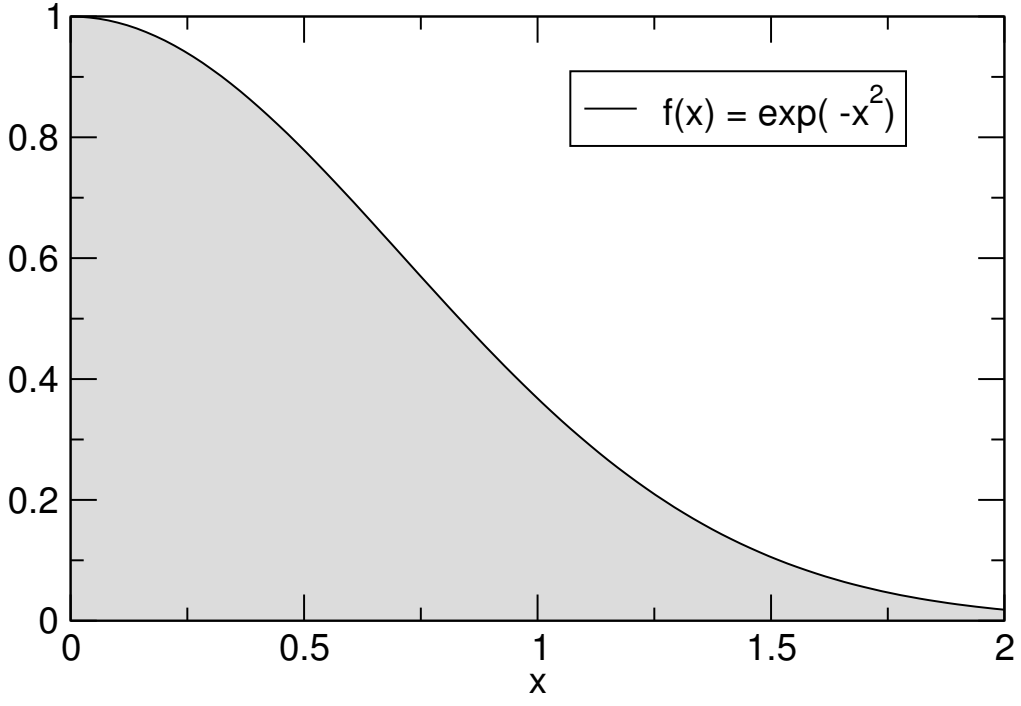
$$y' \equiv \frac{\mathrm{d}y}{\mathrm{d}x} = f(x, y) \tag{8}$$

- example: $\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{1}{m}(g - cv^2)$
- solution is $y(x): \begin{array}{l} \mathbb{R} \to \mathbb{R} \\ x \mapsto y(x) \end{array}$ (*i.e.* a function)
- right hand side $f(x, y)$ depends on x and on solution y
- Can write (8) formally as $y = \int \frac{\mathrm{d}y}{\mathrm{d}x} \mathrm{d}x = \int f(x, y) \mathrm{d}x$. That's why we "integrate differential equations" to solve them.

Numeric computation of definite integrals

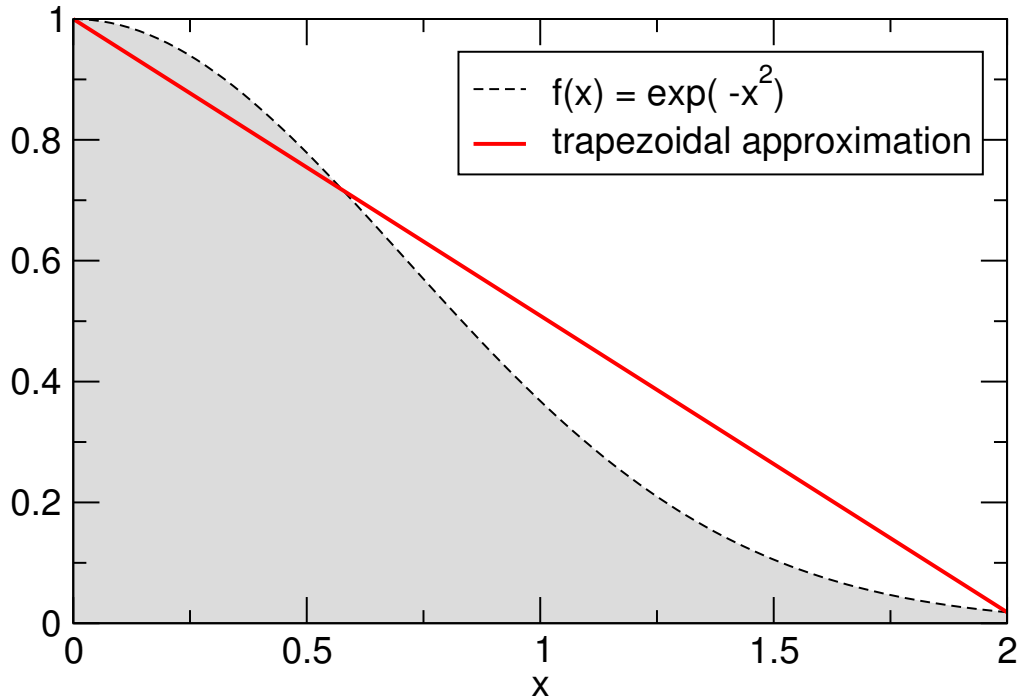
Example:

$$I = \int_0^2 \exp(-x^2)\mathrm{d}x$$



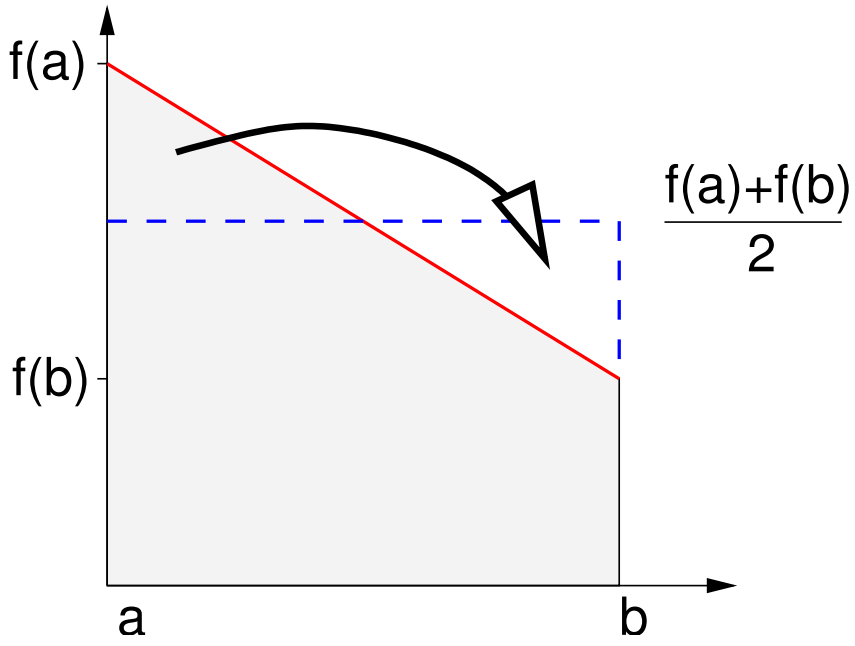
Simple trapezoidal rule

- Approximate function by straight line



Simple trapezoidal rule

- Compute area underneath straight line $p(x)$



- Result

$$A = \int_a^b p(x)\mathrm{d}x = (b-a)\frac{f(a)+f(b)}{2}$$

Example: Simple trapezoidal rule

- Integrate $f(x) = x^2$

$$I = \int_0^2 x^2 \mathrm{d}x$$

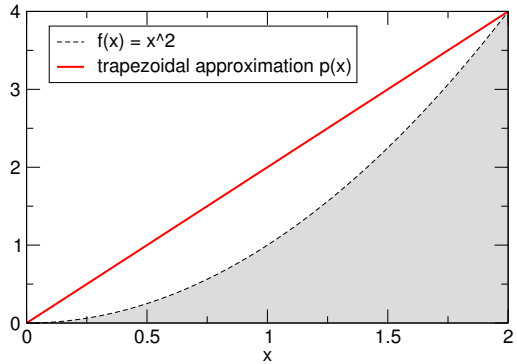
- What is the (correct) analytical answer? Integrating polynomials: $I = \int_a^b x^k \mathrm{d}x = \left[\frac{1}{k+1} x^{k+1}\right]_a^b$
- for $a = 0$ and $b = 2$ and $k = 2$

$$I = \left[\frac{1}{2+1} x^{2+1}\right]_0^2 = \frac{1}{3} 2^3 = \frac{8}{3} \approx 2.6667$$

- Using the trapezoidal rule

$$A = (b-a)\frac{f(a)+f(b)}{2} = 2\frac{0+4}{2} = 4$$

- The correct answer is $I = 8/3$ and the approximation is $A = 4$.
  We thus *overestimate* I by $\frac{A-I}{I} \approx 50\%$.
- Plotting $f(x) = x^2$ together with the approximation reveals why we overestimate I

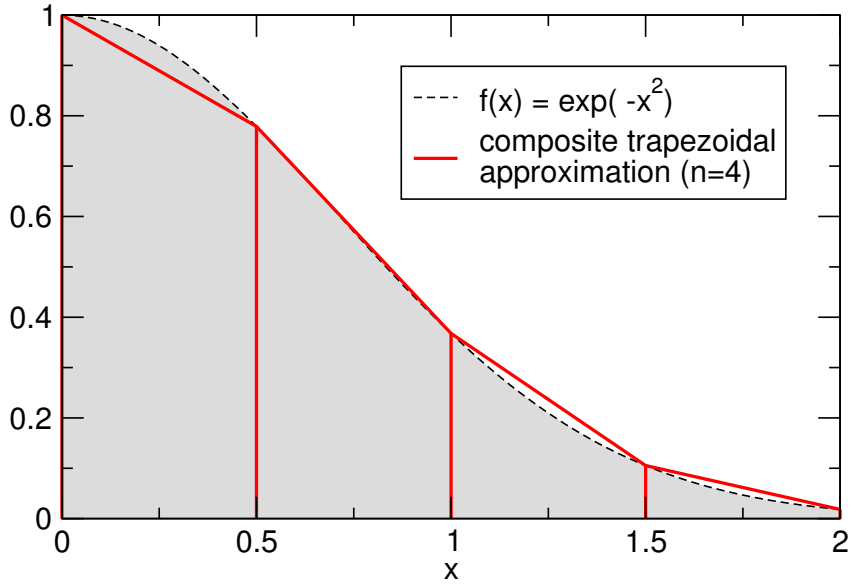


- The linear approximation, $p(x)$, overestimates $f(x)$ everywhere (except at $x = a$ and $x = b$).

Composite trapezoidal rule

Example $f(x) = \exp(-x^2)$:

$$I = \int_0^2 f(x)\mathrm{d}x = \int_0^2 \exp(-x^2)\mathrm{d}x$$



$$I = \int_0^{0.5} f(x)\mathrm{d}x + \int_{0.5}^{1} f(x)\mathrm{d}x + \int_{1}^{1.5} f(x)\mathrm{d}x + \int_{1.5}^{2} f(x)\mathrm{d}x$$

General composite trapezoidal rule

For n subintervals the formulae for the composite trapezoidal rule are

$$
\begin{aligned}
h &= \frac{b-a}{n} \\
x_i &= a + ih \quad \text{with} \quad i = 1, \ldots, n-1
\end{aligned}
$$

$$
\begin{aligned}
A &= \frac{h}{2}\Big(f(a) + 2f(x_1) + 2f(x_2) + \ldots \\
&\qquad +2f(x_{n-2}) + 2f(x_{n-1}) + f(b)\Big) \\
&= \frac{h}{2}\Big(f(a) + \sum_{i=1}^{n-1} 2f(x_i) + f(b)\Big)
\end{aligned}
$$

Error of composite trapezoidal rule

How accurate is the approximation?

We would like to know how much the error decreases when we decrease h (by increasing the number of subintervals, n).

For the composite trapezoidal rule it can be shown that:

$$\int_a^b f(x)\mathrm{d}x = \frac{h}{2}\left(f(a) + f(b) + 2\sum_{i=1}^{n-1} f(x_i)\right) + \mathcal{O}(h^2)$$

The symbol $\mathcal{O}(h^2)$ means that the error term is (smaller or equal to an upper bound which is) proportional to h^2:

- If we take 10 times as many subintervals then h becomes 10 times smaller (because $h = \frac{b-a}{n}$) and the error becomes 100 times smaller (because $\frac{1}{10^2} = \frac{1}{100}$).

Error of composite trapezoidal rule, example

- The table below shows how the error of the approximation, A, decreases with increasing n for

$$I = \int_0^2 x^2 \mathrm{d}x.$$

n	h	A	I	$\Delta = A - I$	rel.err.$= \Delta / I$
1	2.000000	4.000000	2.666667	1.333333	50.0000%
2	1.000000	3.000000	2.666667	0.333333	12.5000%
3	0.666667	2.814815	2.666667	0.148148	5.5556%
4	0.500000	2.750000	2.666667	0.083333	3.1250%
5	0.400000	2.720000	2.666667	0.053333	2.0000%
6	0.333333	2.703704	2.666667	0.037037	1.3889%
7	0.285714	2.693878	2.666667	0.027211	1.0204%
8	0.250000	2.687500	2.666667	0.020833	0.7813%
9	0.222222	2.683128	2.666667	0.016461	0.6173%
10	0.200000	2.680000	2.666667	0.013333	0.5000%
50	0.040000	2.667200	2.666667	0.000533	0.0200%
100	0.020000	2.666800	2.666667	0.000133	0.0050%

- The accuracy we actually require depends on the problem under investigation – no general statement is possible.

Summary trapezoidal rule for numerical integration

- Aim: to find an approximation of

$$I = \int_a^b f(x)\mathrm{d}x$$

- Simple trapezoidal method:
  - approximate $f(x)$ by a simpler (linear) function $p(x)$ and
  - integrate the approximation $p(x)$ exactly.
- Composite trapezoidal method:
  - divides the interval $[a, b]$ into n equal subintervals
  - employs the simple trapezoidal method for each subinterval
  - has an error term of order h^2.

Numerical Integration II

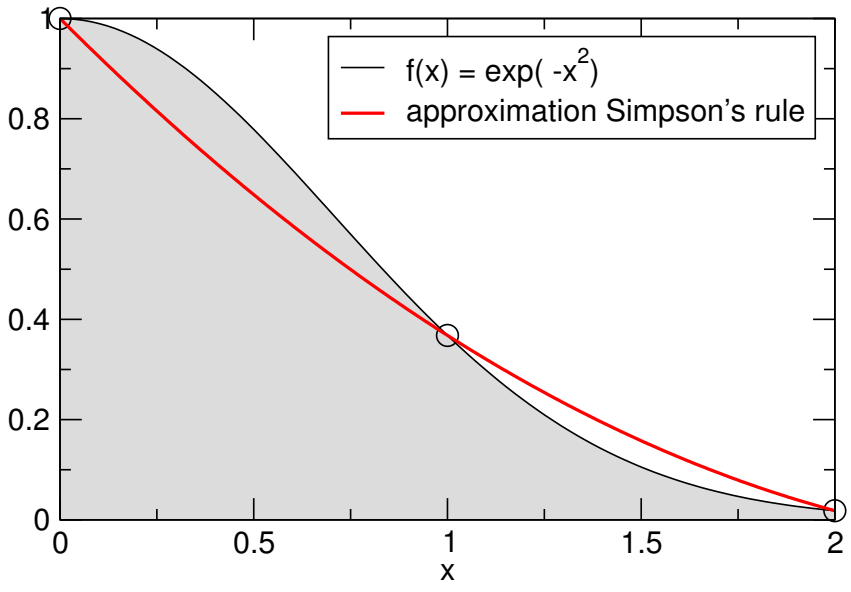
Overview

- Newton-Cotes formulae
- Adaptive methods
- Numerical integration in standard packages
- An interesting idea: Monte-Carlo methods

Simpson's rule [p_2]

Aim: integrate $\int_a^b f(x)\mathrm{d}x$

- approximate $f(x)$ with polynomial $p_2(x)$ of degree 2 (*i.e.* a parabola)
- Need 3 points, choose $a = 0, \quad x_1 = \frac{b+a}{2} = 1, \quad b = 2$:



- This defines a_0, a_1 and a_2 in $p_2(x) = a_0 + a_1 x + a_2 x^2$.

Simpson's rule [p_2]

- Integrate polynomial $p_2(x)$

$$A = \int_a^b p_2(x)\mathrm{d}x = \int_a^b a_0 + a_1 x + a_2 x^2 \mathrm{d}x = \ldots$$

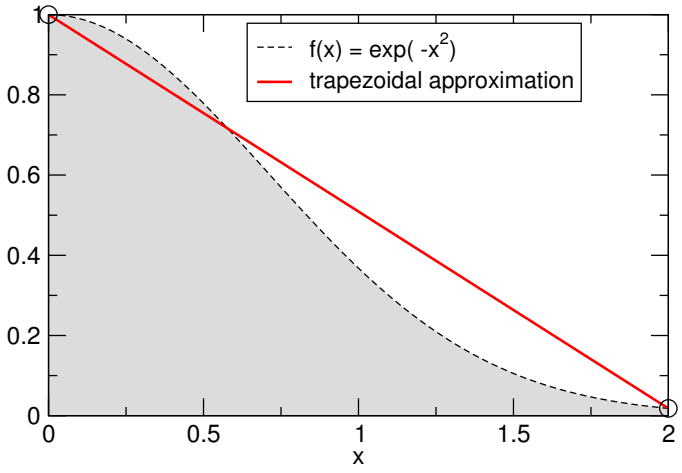
  - substitute in a_0, a_1 and a_2
    - a_0, a_1 and a_2 depend on $a, x_1, b, f(a), f(x_1), f(b)$
    - assume we know what they are
      ($\rightarrow$ Lagrange interpolation polynomial)
  - After expanding and summarising we find:

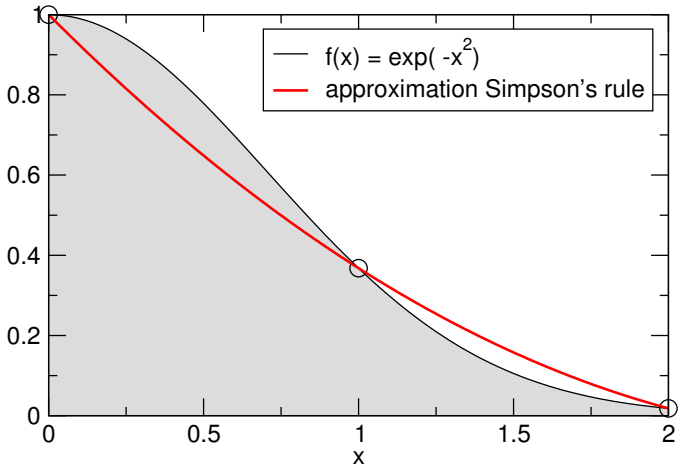
$$\ldots = \frac{b-a}{6}\left(f(a) + 4f(x_1) + f(b)\right)$$

- The rule of Simpson: assume $f(x) \approx p_2(x)$

$$I = \int_a^b f(x)\mathrm{d}x \approx \int_a^b p_2(x)\mathrm{d}x = \frac{b-a}{6}\left(f(a) + 4f(x_1) + f(b)\right)$$

Comparison trapezoidal rule / Simpson's rule



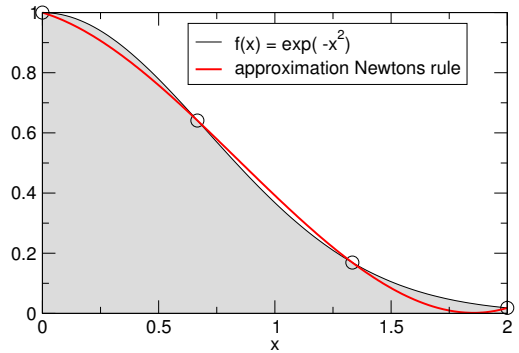


- Trapezoidal rule (left)
  - approximate $f(x)$ with a polynomial of degree 1 (*i.e.* a linear function)
  - need 2 function evaluations (at a and b)
- Simpson's rule (right)
  - approximate $f(x)$ with a polynomial of degree 2 (*i.e.* a parabolic function)
  - need 3 function evaluations (at a and b and at $(b-a)/2$)
- Simpson's rule can be shown to be (much) more accurate

Newton's rule [p_3]

Aim: integrate $\int_a^b f(x)\mathrm{d}x$

- approximate $f(x)$ with polynomial $p_3(x)$ of degree 3
- Need 4 equidistant points
  $a, x_1 = a + \frac{(b-a)}{3}, x_2 = a + 2\frac{(b-a)}{3}, b$



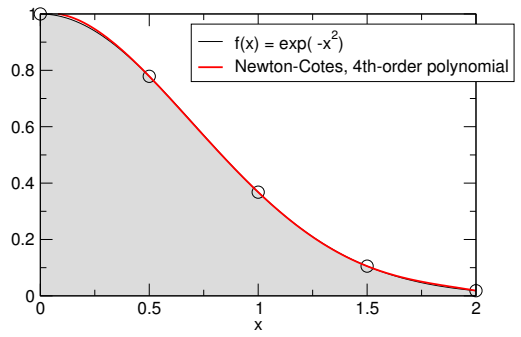
- Newton's rule:

$$A = \int_a^b p_3(x)\mathrm{d}x = \frac{b-a}{8}(f(a) + 3f(x_1) + 3f(x_2) + f(b))$$

Bode's rule

Aim: integrate $\int_a^b f(x)\mathrm{d}x$

- approximate $f(x)$ with polynomial $p_4(x)$ of degree 4
- Need 5 equidistant points $a, x_1 = a + \frac{(b-a)}{4}$, $x_2 = a + 2\frac{(b-a)}{4}$, $x_3 = a + 3\frac{(b-a)}{4}$, b

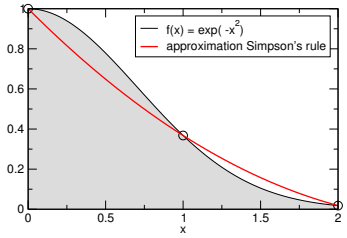


- Bode's rule:

$$\begin{aligned} A &= \int_a^b p_3(x)\mathrm{d}x \\ &= \frac{b-a}{90}\left(7f(a) + 32f(x_1) + 12f(x_2) + 32f(x_3) + 7f(b)\right) \end{aligned}$$

Newton-Cotes formulae

Aim: integrate $I = \int_a^b f(x)\mathrm{d}x$



- evaluate $f(x)$ at $k+1$ equidistant points between a and b
- approximate $f(x)$ by polynomial $p_k(x)$ of degree k
- integrate $p_k(x)$ exactly to approximate I
- Examples
  - k=1: trapezoidal rule (linear approximation $p_1(x)$)
  - k=2: Simpson's rule (parabolic approximation $p_2(x)$)
  - k=3: Newton's rule (cubic approximation $p_3(x)$)
  - k=4: Bode's rule (quartic approximation $p_4(x)$)

Error of composite Newton-Cotes formulae

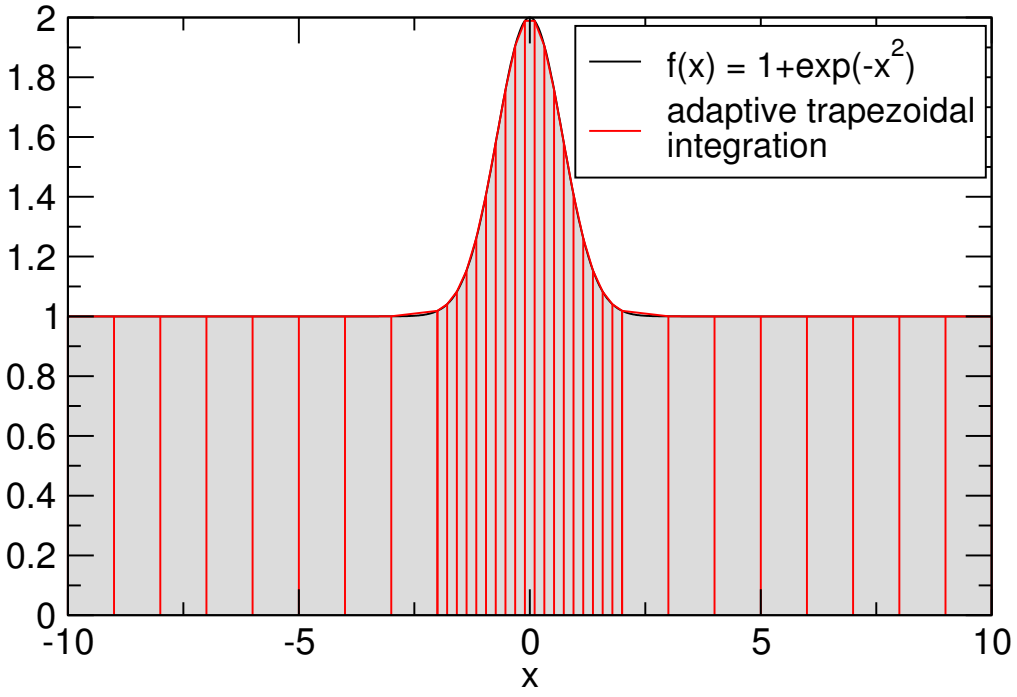
Name	approximate $f(x)$ with	error term
trapezoidal	$p_1(x)$	$\mathcal{O}(h^2)$
Simpson's	$p_2(x)$	$\mathcal{O}(h^4)$
Newton's	$p_3(x)$	$\mathcal{O}(h^4)$
Bode's	$p_4(x)$	$\mathcal{O}(h^6)$
?	$p_5(x)$	$\mathcal{O}(h^6)$

- Interesting behaviour of error terms (need further mathematics to understand this)
- Composite Simpson's rule is commonly used
  - fairly simple
  - but already $\mathcal{O}(h^4)$

Summary Newton-Cotes formulae

- closed
  - trapezoidal rule (p_1)
  - Simpson's rule (p_2)
  - Newton's rule (p_3)
  - Bode's rule (p_4)
- open
  - midpoint-rule
- composite versions

Adaptive integration



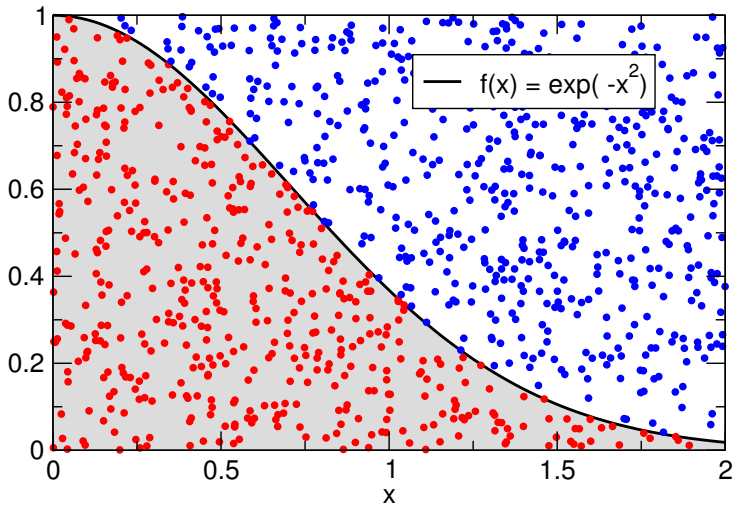
- can improve efficiency of numerical integration by
  - using a small step size h where $f(x)$ varies quickly
  - using large steps h where $f(x)$ varies slowly
- requires the ability to estimate the error in each step

Numerical integration in standard packages

- Numerical integration available, for example in Matlab, Scientific Python, Maple, ...
- Often, the command is called `quad`. (`quad` stands for QUADrature — a (old) term for integration.)
- For example, Matlab's `quad` command is based on a
  - composite
  - Simpson rule with
  - adaptive step size.
- The available integration methods are usually very sophisticated and sufficient for our needs
- You can integrate functions in Python using `scipy.integrate.quad()`.

Monte-Carlo Integration

- Strategy: Use random numbers to estimate an area underneath a function
  - create many random points in rectangle
  - compute area R of rectangle (simple)
  - compute fraction f of points being underneath the curve
  - approximate integral with fR.
- Example: 1000 points, 461 underneath curve, $R = 2 \Rightarrow A \approx 0.92$ (relative error of 5%)



- Use Monte-Carlo integration
  - for high dimensional integrals $\int\int\int f(x, y, z)\mathrm{d}x\mathrm{d}y\mathrm{d}z$
  - for computing the volume of complex shapes

Summary Integration 2

- Newton-Cotes formulae, simple and composite
  - trapezoidal and Simpson's rule
- adaptive integration
- numerical integration in standard packages
- Monte-Carlo integration

Tricks we have seen

- can't integrate $f(x)$? $\rightarrow$ Then approximate $f(x)$ with simple function and integrate approximation
- parts of a function may change slowly, other parts rapidly $\rightarrow$ adaptive integration
- can use a completely different approach $\rightarrow$ Monte-Carlo integration

Ordinary Differential Equations (ODEs)

Ordinary Differential Equations

- Many processes, in particular *time-dependent* processes, can be described as Ordinary Differential Equations (ODEs), such as dynamics of engineering systems, quantum physics, chemical reactions, biological systems modelling, and population dynamics.
- ODEs have *exactly one* independent variable t (often, but not always representing time).
- The simplest ODE has one degree of freedom y.
- The *solution* of the ODE is the function $y(t)$. Examples:
  - temperature as a function of time
  - distance a car has moved as function of time
  - population of species as function time

Ordinary Differential Equations

- We are typically being given
  - an initial value y_0 of $y(t)$ at some time t_0 and
  - the ODE itself which relates the change of y with t to some function $f(t, y)$, i.e.

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y) \tag{9}$$

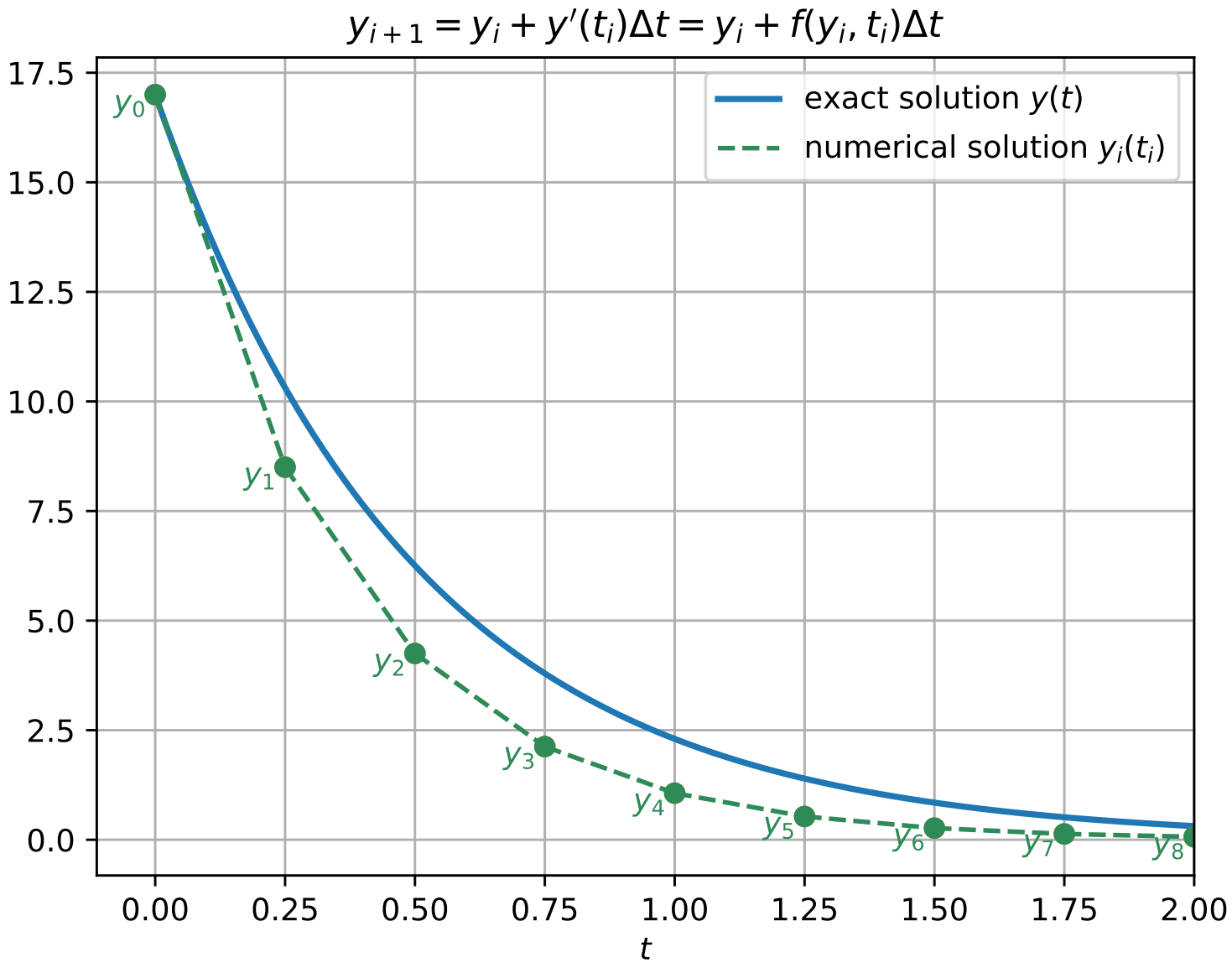
- Example: looking for solution $y(t)$ from $t_0 = 0$ to $t = 2$ of

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -2y \quad \text{with} \quad y_0 = y(t_0) = 17$$

The exact solution is $y(t) = 17\,\mathbf{exp}(-2t)$.

- In general, a vector $\mathbf{y}$ with k components can depend on the independent variable t: this is a *system* of ordinary differential equations with k degrees of freedom.

Principle of finding numerical solution to ODE



Interface `solve_ivp`

- aim: solve

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(t, y)$$

- `from scipy.integrate import solve_ivp`
- `solve_ivp` has the following input and output parameters:

```
sol = solve_ivp(f, t_span, y0)
```

Input:

  - `f` is function `f(t, y)` that returns the right-hand side
  - `t_span` is a tuple `(t0, tf)` describing the span of t for which we search the solution
  - `y0` is the initial value of the solution at time t_0 (i.e. $y_0 = y(t_0)$)

Output:

  - `sol` is a `OdeResult` object that contains the solution

Using `solve_ivp` – example 1

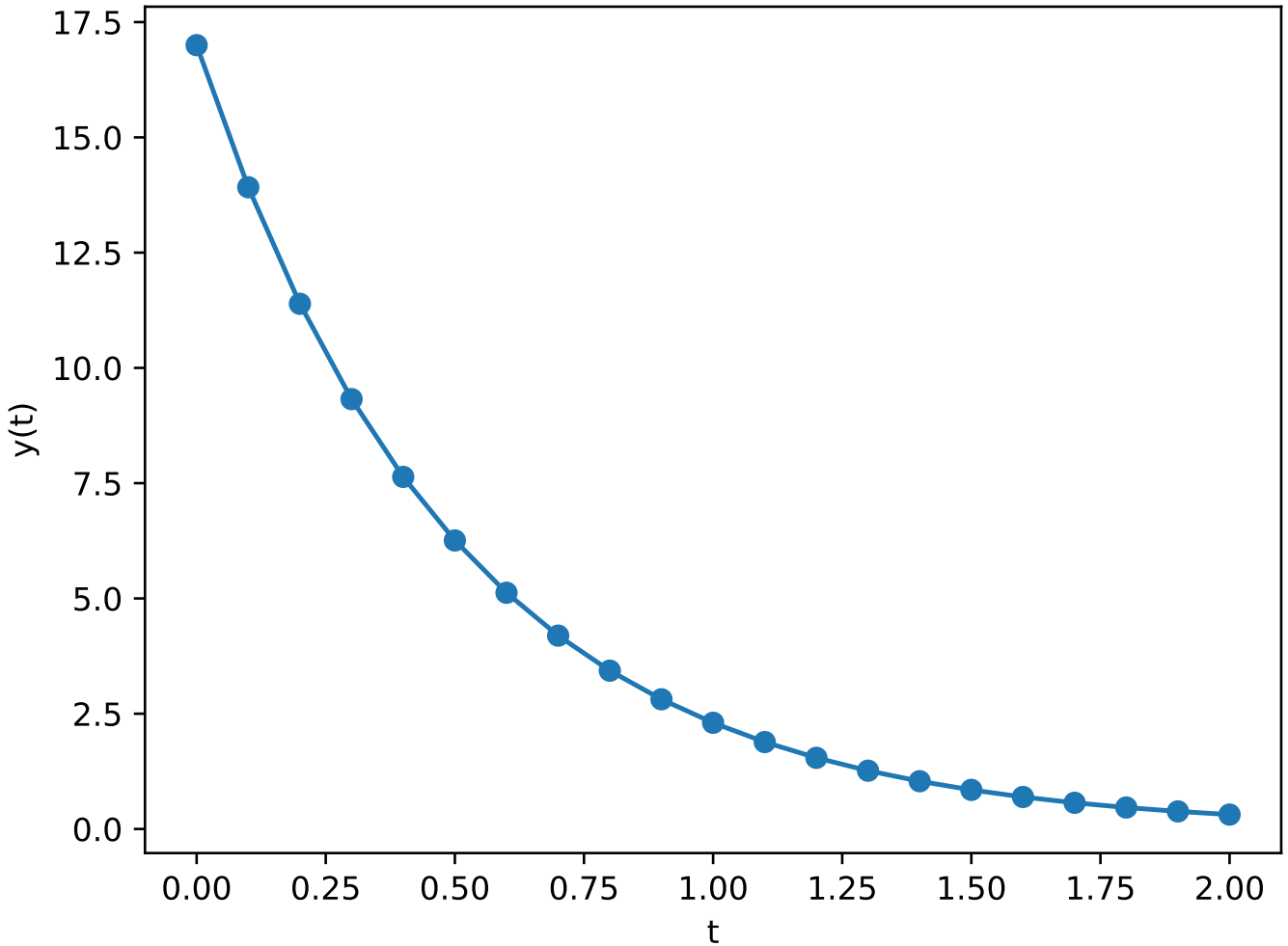
Require solution $y(t)$ from $t = 0$ to $t = 2$ of

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -2y \quad \text{with} \quad y(0) = 17$$

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

def f(t, y):
    """this is the rhs of the ODE to integrate, i.e. dy/dt=f(y,t)"""
    return -2 * y

y0 = [17]  # initial value y0=y(t0)
t0 = 0  # integration limits for t: start at t=0
tf = 2  # and finish at t=2
t_eval = np.linspace(t0, tf, 21)

sol = solve_ivp(fun=f, t_span=[t0, tf], y0=y0, t_eval=t_eval)

fig, ax = plt.subplots()
ax.plot(sol.t, sol.y[0], "o-"); ax.set_xlabel("t");
```

Using `solve_ivp` – example 1, solution

Solution:



Using `solve_ivp` – example 2

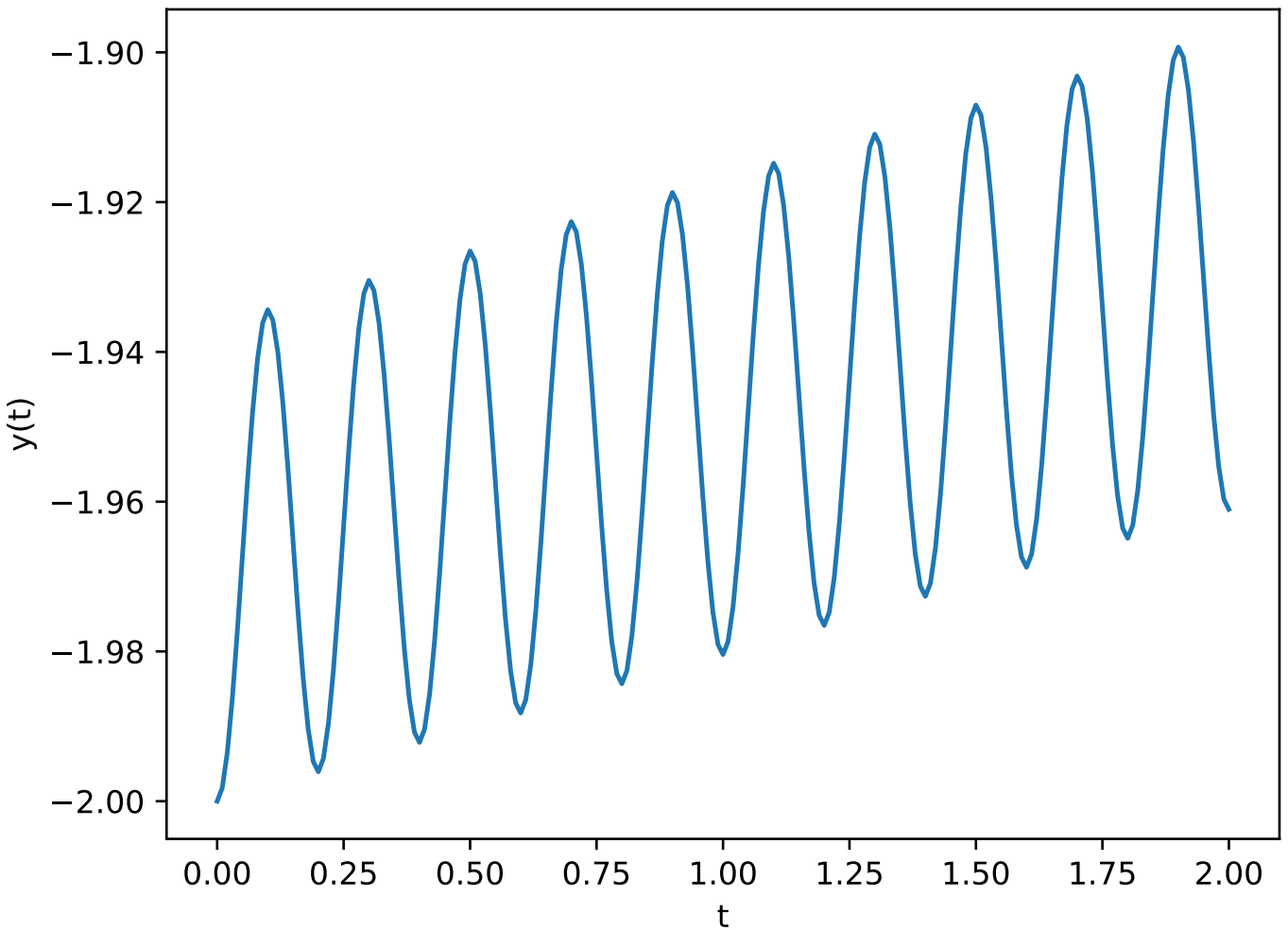
Require solution $y(t)$ from $t = 0$ to $t = 2$ of

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -\frac{1}{100}y + \sin(10\pi t) \quad \text{with} \quad y(0) = -2$$

```
import math
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import solve_ivp

def f(t, y):
    return -0.01 * y + math.sin(10 * math.pi * t)

ts = np.arange(0, 2.01, 0.01)
y0 = [-2]
sol = solve_ivp(f, t_span=(0, 2), y0=y0,
                t_eval=ts, atol=1e-8, rtol=1e-8)

fig, ax = plt.subplots()
ax.plot(sol.t, sol.y[0])
ax.set_xlabel("t"); ax.set_ylabel("y(t)")
fig.savefig("odeintexample2.pdf")
```

Using `solve_ivp` – example 2, solution

Solution:



2nd order ODE

- Any second order ODE can be re-written as two coupled first order ODE
- Example: Harmonic Oscillator (HO)
  - Differential equation $\frac{\mathrm{d}^2 r}{\mathrm{d}t^2} = -\omega^2 r$ or short $r'' = -\omega^2 r$
  - Introduce $v = r'$
  - rewrite equation as two first order equations

$$r'' = -\omega^2 r \qquad \longrightarrow \qquad \begin{aligned} v' &= -\omega^2 r \\ r' &= v \end{aligned}$$

- General strategy:
  - convert higher order ODE into a set of (coupled) first order ODE
  - use computer to solve set of 1st order ODEs

2nd order ODE – using `solve_ivp`

- One 2nd order ODE $\rightarrow$ 2 coupled 1st order ODEs
- Integration of *system* of 1st order ODEs:
  - "pretty much like integrating one 1st order ODE" but
  - y is now a vector (and so is f):

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = \mathbf{f}(t, \mathbf{y}) \quad \Longleftrightarrow \quad \begin{pmatrix} \frac{\mathrm{d}y_1}{\mathrm{d}t} \\ \frac{\mathrm{d}y_2}{\mathrm{d}t} \end{pmatrix} = \begin{pmatrix} f_1(t, \mathbf{y}) \\ f_2(t, \mathbf{y}) \end{pmatrix}$$

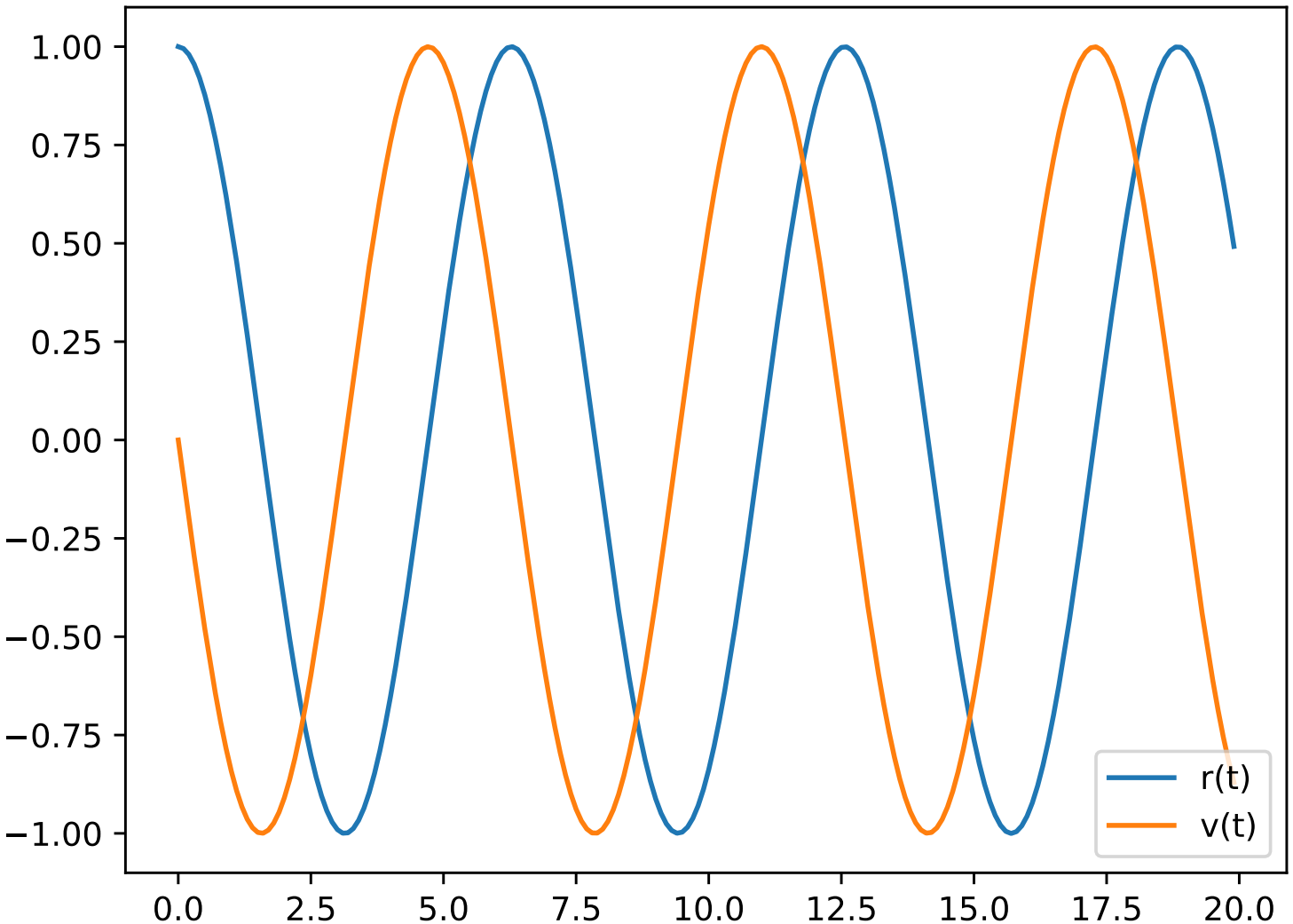
  - need to pack and unpack variables into the *state vector* $\mathbf{y}$:
  - Example harmonic oscillator:
    - decide to use this packing: $\mathbf{y} = (r, v)$
    - then $\mathbf{f}$ needs to return $\mathbf{f} = \left(\frac{\mathrm{d}r}{\mathrm{d}t}, \frac{\mathrm{d}v}{\mathrm{d}t}\right)$
  - the `sol` object returned by `solve_ivp` has an attribute `sol.y` which contains a vector $\mathbf{y}$ for every time step
    - need to extract results for r and v from that matrix $\rightarrow$ see next slide

2nd order ODE – Python solution harmonic oscillator (HO)

```
from numpy import array, arange
from scipy.integrate import solve_ivp

def f(t, y):                # right hand side, takes array(!) y
    omega = 1
    r = y[0]                # extract r from array y
    v = y[1]                # extract v from array y
    drdt = v                # compute right hand side
    dvdt = -omega ** 2 * r
    return array([drdt, dvdt])  # return array

ts = arange(0, 20, 0.1)    # required times for solution
r0 = 1                     # initial r
v0 = 0                     # initial v
y0 = [r0, v0]              # combine r and v into y

sol = solve_ivp(f, (0, 20), y0, t_eval=ts)    # solve ODEs

rs = sol.y[0]              # extract result: r(t)
vs = sol.y[1]              # extract result: v(t)
```

2nd order ODE – result

Solution



Summary 2nd order system

- Strategy:
  - transform one 2nd order ODE into 2 (coupled) first order ODEs
  - solve both first order ODEs simultaneously
- Need to use vectors (typically “arrays”) in to pass state vector to right-hand-side function.
- Use example on previous slides as guidance.

2 Coupled ODEs: Predator-Prey problem

- Predator and prey. Let
  - $p_1(t)$ be the number of rabbits
  - $p_2(t)$ be the number of foxes
- Time dependence of p_1 and p_2:
  - Assume that rabbits proliferate at a rate a. Per unit time a number ap_1 of rabbits is born.
  - Number of rabbits is reduced by collisions with foxes. Per unit time cp_1p_2 rabbits are eaten.
  - Assume that birth rate of foxes depends only on food intake in form of rabbits.
  - Assume that foxes die a natural death at a rate b.
- Numbers
  - rabbit birth rate $a = 0.7$

2 Coupled ODEs: Predator-Prey problem

  - rabbit-fox-collision rate $c = 0.007$
  - fox death rate $b = 1$
- Put all together in predator-prey ODEs

$$\begin{aligned} p_1' &= ap_1 - cp_1p_2 \\ p_2' &= cp_1p_2 - bp_2 \end{aligned}$$

- Solve for $p_1(0) = 70$ and $p_2(0) = 50$ for 30 units of time:

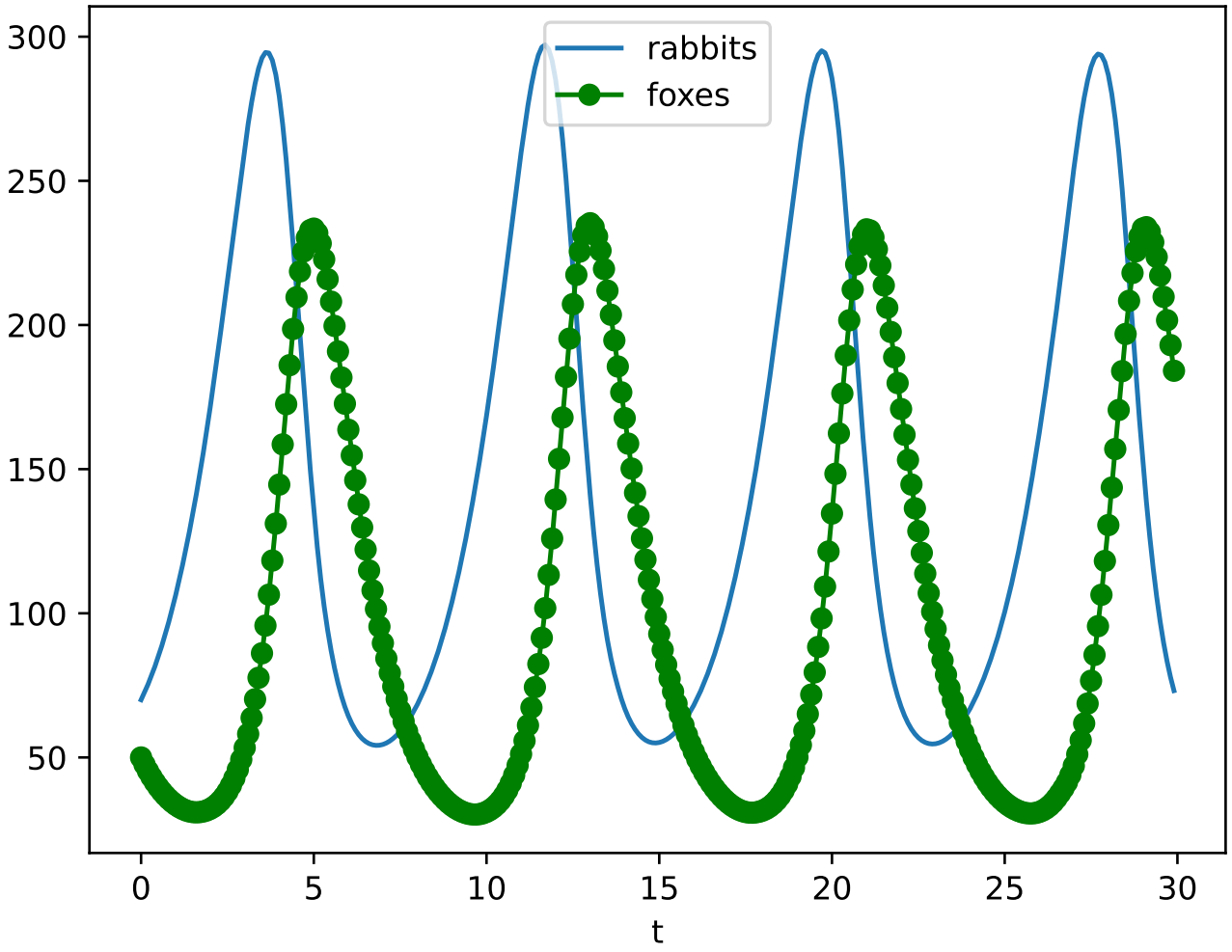
2 Coupled ODEs: Predator-Prey problem

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  from scipy.integrate import solve_ivp
4
5  def rhs(t, y):
6      a = 0.7
7      c = 0.007
8      b = 1
9      p1 = y[0]
10     p2 = y[1]
11
12     dp1dt = a * p1 - c * p1 * p2
13     dp2dt = c * p1 * p2 - b * p2
14
15     return np.array([dp1dt, dp2dt])
16
17 p0 = [70, 50]       # initial condition
18 t0 = 0
19 tfinal = 30
20 ts = np.arange(t0, tfinal, 0.1)
```

2 Coupled ODEs: Predator-Prey problem

```
21
22  sol = solve_ivp(rhs, [t0, tfinal], p0, t_eval=ts)
23
24  p1 = sol.y[0]                  # extract p1 and
25  p2 = sol.y[1]                  # p2
26
27  fig, ax = plt.subplots()
28  ax.plot(sol.t, p1, label='rabbits')
29  ax.plot(sol.t, p2, '-og', label='foxes')
30  ax.legend()
31  ax.set_xlabel('t')
32  fig.savefig('predprey.pdf')
```

2 Coupled ODEs: Predator-Prey problem



Outlook

Suppose we want to solve a (vector) ODE based on Newton's equation of motion in three dimensions:

$$\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = \frac{\mathbf{F}(\mathbf{r}, \mathbf{v}, t)}{m}$$

Rewrite as two first order (vector) ODEs:

$$\begin{aligned} \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} &= \frac{\mathbf{F}(\mathbf{r}, \mathbf{v}, t)}{m} \\ \frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t} &= \mathbf{v} \end{aligned}$$

Need to pack 6 variables into "$\mathbf{y}$": for example

$$\mathbf{y} = (r_x, r_y, r_z, v_x, v_y, v_z)$$

Right-hand-side function $\mathbf{f}(\mathbf{y}, t)$ needs to return:

$$\mathbf{f} = \left(\frac{\mathrm{d}r_x}{\mathrm{d}t}, \frac{\mathrm{d}r_y}{\mathrm{d}t}, \frac{\mathrm{d}r_z}{\mathrm{d}t}, \frac{\mathrm{d}v_x}{\mathrm{d}t}, \frac{\mathrm{d}v_y}{\mathrm{d}t}, \frac{\mathrm{d}v_z}{\mathrm{d}t} \right) \tag{10}$$

Outlook examples

- Example: Molecular dynamics simulations have one set of 6 degrees of freedom as in equation (10) *for every atom* in their simulations.
- Example: Material simulations discretise space into finite elements, and for dynamic simulations the number of degrees of freedom are proportional to the number of nodes in the mesh.
- Very sophisticated time integration schemes for ODEs available (such as ”sundials” suite).
- The tools in `scipy.integrate` are pretty useful already (`solve_ivp` has multiple solvers - we have only used the default Runge Kutta 45 solver.).

Symbolic Python (sympy)

Symbolic Python (`sympy`)

What?

- symbolic algebra - computing with variables not numbers (like Mathematica, SageMath, Wolfram Alpha, other, ...)

Why?

- Use symbolic computation before moving to numerical calculations to avoid mistakes
- and to simplify expression as much as possible.
- Write computer code (or LaTeX) automatically from sympy
- Or use from Python using `sympy.lambdify`

Why symbolic python?

- sympy is not the only option - other packages may well be faster/know more mathematics, but
  - sympy connects well to Python ecosystem of computational science tools
  - free and open source
  - scriptable: can integrate into *automatic* workflows
  - very powerful

Symbolic Python - basics

```
>>> import sympy
>>> x = sympy.Symbol('x')    # define symbolic
>>> y = sympy.Symbol('y')    # variables
>>> x + x
2*x
>>> t = (x + y)**2
>>> print(t)
(x + y)**2
>>> sympy.expand(t)
x**2 + 2*x*y + y**2
>>> sympy.pprint(t)          # PrettyPRINT
       2
(x + y)
>>> sympy.printing.latex(t) # Latex export
'\\left(x + y\\right)^{2}'
```

Substituting values and numerical evalution

```
>>> t
(x + y)**2
>>> t.subs(x, 3)                # substituting variables
(y + 3)**2                      # or values
>>> t.subs(x, 3).subs(y, 1)
16
>>> n = t.subs(x, 3).subs(y, sympy.pi)
>>> print(n)
(3 + pi)**2
>>> n.evalf()                   # EVALuate to Float
37.7191603226281
>>> p = sympy.pi
>>> p
pi
>>> p.evalf()
3.14159265358979
>>> p.evalf(47)                 # request 47 digits
3.1415926535897932384626433832795028841971693993
```

Working with infinity

```
>>> from sympy import limit, sin, oo
>>> limit(1/x, x, 50)        # what is 1/x if x --> 50
1/50
>>> limit(1/x, x, oo)        # oo is infinity
0
>>> limit(sin(x) / x, x, 0)
1
>>> limit(sin(x)**2 / x, x, 0)
0
>>> limit(sin(x) / x**2, x, 0)
oo
```

Taylor series

```
>>> from sympy import series
>>> taylorseries = series(sin(x), x, 0)
>>> taylorseries
x - x**3/6 + x**5/120 + O(x**6)
>>> sympy.pprint(taylorseries)
     3     5
    x     x
x - -- + --- + O(x**6)
    6    120
>>> taylorseries = series(sin(x), x, 0, n=10)
>>> sympy.pprint(taylorseries)
     3     5     7       9
    x     x     x       x
x - -- + --- - ---- + ------ + O(x**10)
    6    120   5040   362880
```

Integration

```
>>> from sympy import integrate
>>> a, b = sympy.symbols('a, b')
>>> integrate(2*x, (x, a, b))
-a**2 + b**2
>>> integrate(2*x, (x, 0.1, b))
b**2 - 0.01
>>> integrate(2*x, (x, 0.1, 2))
3.99000000000000
```

Solving equations

Finally, we can solve non-linear equations, for example:

```
>>> (x + 2)*(x - 3)     # define quadratic equation
                        # with roots x=-2, x=3
(x - 3)*(x + 2)
>>> r = (x + 2)*(x - 3)
>>> r.expand()
x**2 - x - 6
>>> sympy.solve(r, x)   # solve r = 0
[-2, 3]                 # solution is x = -2, 3
```

Lambdify sympy expressions

```
>>> from sympy import sin, cos, symbols, lambdify
>>> import numpy as np
>>> x = symbols('x')
>>> symb = sin(x) + cos(x)
>>> symb
sin(x) + cos(x)
>>> f = lambdify(x, symb, 'numpy')
>>> f(0)
1.0
>>> f(np.linspace(0, 1, 10))
array([1.        , 1.10471614, 1.19580783, 1.27215164,
1.33280603, 1.37702295, 1.40425706, 1.4141725 ,
1.40664697, 1.38177329])
```

Workflow: Create sympy expressions, then lambdify them to execute faster.

Sympy summary

- Sympy is purely Python based
- fairly powerful (although better open source tools are available if required)
- we should use computers for symbolic calculations routinely alongside pen and paper, and numerical calculations
- can produce LaTeX output
- can produce C and Fortran code (and wrap this up as a Python function automatically ("autowrap"))

APPENDIX

Practical computational science recommendations

Research software development

- use version control
- start in Python
- use tests
- keep it simple
- make it readable
- use notebooks for examples -> documentation (sphinx)
- if you need to change/extend/rewrite software
  - automatic tests are your friend
  - continuous integration
  - we can also *test documentation* (nbval, doctest)

Software too slow?

- Identify where it is slow (”Profiling”)
- move execution of ’slow’ operations to compiled code where necessary
  - through use of numpy
  - through use of Cython
  - through linking to compiled code (ctypes, cython, swig, boost, f2py, ...) to talk to C, C++, Fortran, Rust, OCaml, ...
- Parallelise through use of libraries that can execute in parallel
  - mkl numpy
  - dask
  - numba
  - pytorch, cupy, jax, ...

  Includes use of GPUs.

Good practice Computational Science

- use notebooks to document computational work
- use version control (for software, reports and papers)
- archive software and notebooks (publicly if you can)
- in particular for (more reproducible) publications [1]:
  - publish git repo with paper (Zenodo?)
  - document your software environment
  - if you can create it automatically, this is best
  - consider making your repo `binder`-enabled ()

[1] Beg, Fangohr, etal: *Using Jupyter for reproducible scientific workflows*, Computing in Science and Engineering 23, 36-46 10.1109/MCSE.2021.3052101 (2021)

Give back to the community where you can

- Contribute to the open source tools you are using, for example
  - provide bug reports
  - suggest improvements to documentation
  - make feature requests
  - helping other users
  - ...
- Cite software that is important for your work in your papers: many packages suggest what to cite if you use them

Useful tools

Black — The Uncompromising Code Formatter

"one style, as long as it is this one"

- leave formatting to `black`
- focus on content (rather than formatting)
- makes code review easier
- compatible with PEP8

Usage:

- Check if `file.py` sticks to Black standard:

```
black --check file.py
```

- Autoformat `file.py`:

```
black file.py
```

mypy and pytype — static type checking

Code with type annotations (see slide 339) can be analysed statically (i.e. without being executed).

Important tools:

- `mypy` `https://mypy-lang.org/`
- `pytype` `https://github.com/google/pytype`

Pytype can also infer types (to some degree) and merge to source.

A gentle introduction to the topic in Talk Python to Me podcast, episode 151.

*Integer division in Python 2 and 3

Python 2: Integer division

Dividing two integers in Python 1 and 2 returns an integer:

```
>>> 1 / 2
0                    # might have expected 0.5, not 0
```

We find the same behaviour in Java, C, Fortran, and many other programming languages.

Solutions:

- change (at least) one of the integer numbers into a floating point number (i.e. `1` $\rightarrow$ `1.0`).

Python 2: Integer division

```
>>> 1.0 / 2
0.5
```

- Or use `float` function to convert variable to float

```
>>> a = 1
>>> b = 2
>>> 1 / float(b)
0.5
```

Python 2: Integer division

- Or make use of Python's future division:

```
>>> from __future__ import division
>>> 1 / 2
0.5
```

Python 3: Integer division

In Python 3:

```
>>> 1 / 2
0.5
```

Dividing 2 integers returns a float:

```
>>> 4 / 2
2.0
>>> type(4 / 2)
<class float>
```

Python 3: Integer division

If we want integer division (i.e. an operation that returns an integer, and/or which replicates the default behaviour of Python 2), we use `//`:

```
>>> 1 // 2
0
```

Legacy string formatting

String formatting method overview

"f-strings": most convenient and recommended method (2016, see slide 163):

```
>>> 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'
```

String formatting overview history

- 1. 1991: `%` operator (Python 1 and 2)
- 2. 2006: `str.format()` “new style“ or “advanced” string formatting (Python 3)
- 3. 2016: f-strings (Python 3.6)

1. String formatting: the percentage (%) operator

% operator syntax

Syntax: `A % B`

where `A` is a string, and `B` a Python object, or a tuple of

Python objects.

The format string `A` needs to contain *k* format specifiers if the tuple has length *k*. The operation returns a string.

Example: basic formatting of one number

1. String formatting: the percentage (%) operator

```
>>> import math
>>> p = math.pi
>>> "%f" % p          # format p as float (%f)
'3.141593'            # returns string
>>> "%d" % p          # format p as integer (%d)
'3'
>>> "%e" % p          # format p in exponential style
'3.141593e+00'
>>> "%g" % p          # format using fewer characters
'3.14159'
```

1. String formatting: the percentage (%) operator

The format specifiers can be combined with arbitrary characters in string:

```
>>> 'the value of pi is approx %f' % p
'the value of pi is approx 3.141593'
>>> '%d is my preferred number' % 42
'42 is my preferred number'
```

Combining multiple objects

1. String formatting: the percentage (%) operator

```
>>> "%d times %d is %d" % (10, 42, 10 * 42)
'10 times 42 is 420'
>>> "pi=%f and 3*pi=%f is approx 10" % (p, 3*p)
'pi=3.141593 and 3*pi=9.424778 is approx 10'
```

Fixing width and/or precision of resulting string

Fixing width and/or precision of resulting string

```
>>> '%f' % 3.14      # default width and precision
'3.140000'

>>> '%10f' % 3.14    # 10 characters long
'  3.140000'

>>> '%10.2f' % 3.14 # 10 long, 2 post-dec digits
'      3.14'

>>> '%.2f' % 3.14    # 2 post-decimal digits
'3.14'

>>> '%.14f' % 3.14  # 14 post-decimal digits
'3.14000000000000'
```

Common formatting specifiers

A list of common formatting specifiers, with example output for the astronomical unit (AU) which is the distance from Earth to Sun [in metres]:

```
>>> AU = 149597870700   # astronomical unit [m]
>>> "%f" % AU           # line 1 in table
'149597870700.000000'
```

specifier	style	Example output for AU
%f	floating point	149597870700.000000
%e	exponential notation	1.495979e+11
%g	shorter of %e or %f	1.49598e+11
%d	integer	149597870700
%s	str()	149597870700
%r	repr()	149597870700

Summary %-operator for printing

Create string using the %-operator, then pass the string to the print function. Typically done in the same line:

```
>>> import math
>>> print("My pi = %.2f." % math.pi)
My pi = 3.14.
```

Print multiple values:

```
>>> print("a=%d b=%d" % (10, 20))
a=10 b=20
```

Summary %-operator for printing

Very similar syntax exists in other languages, for example C and Matlab, for formatted data output to screen and files.

2. New style string formatting (`format` method)

A new(er) system of built-in formatting has been proposed (PEP3101), titled Advanced String Formatting and is available in Python 3.

Basic ideas in examples:

- Pairs of curly braces are the placeholders.

```
>>> "{} owns {} bikes".format('Peter', 4)
'Peter owns 4 bikes'
```

2. New style string formatting (`format` method)

- Formatting behaviour of `%f` can be achieved through `{:f}`, (same for `%d`, `%e`, etc)

```
>>> "Pi is approx {:f}.".format(math.pi)
'Pi is approx 3.141593.'
```

- Width and post decimal digits can be specified as before:

```
>>> "Pi is approx {:6.2f}.".format(math.pi)
'Pi is approx   3.14.'
>>> "Pi is approx {:.2f}.".format(math.pi)
'Pi is approx 3.14.'
```

2. New style string formatting (`format` method)

Further Reading

- Examples
  http://docs.python.org/library/string.html#format-examples
- Python Enhancement Proposal 3101

3. f-strings (formatted string literals)

- Introduced in Python 3.6
- Described in PEP498
  https://www.python.org/dev/peps/pep-0498/
- combines with `str.format` syntax

f-strings examples

```
>>> name = "Fred"
>>> f"He said his name is {name}."
'He said his name is Fred.'
>>> value = 12.34567
>>> f"result: {value}"
'result: 12.34567'
```

f-strings re-use new style syntax

We can combine f-strings with new format specifiers:

f-strings re-use new style syntax

```
>>> value = 12.34567
>>> f"result: {value:10}"  # 10 spaces
'result:   12.34567'
>>> f"result: {value:e}"   # %e behaviour
'result: 1.234567e+01'
>>> f"result: {value:f}"   # %f behaviour
'result: 12.345670'
>>> f"result: {value:.4f}" # 4 post-decimal digits
'result: 12.3457'
>>> f"result: {value:.4}"  # 4 digits precision
'result: 12.35'
```

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 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'
```

Comparison string formatting generations 1 (repeat slide)

Example 1

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

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

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

Comparison string formatting generations 2

Example 2

```
>>> import math
>>> x = math.pi

# conventional:
>>> "x is %f and x^2 is approx %.1f" % (x, x**2)
'x is 3.141593 and x^2 is approx 9.9'

# new-style:
>>> "x is {:f} and x^2 is approx {:.1f}".format(x, x**2)
'x is 3.141593 and x^2 is approx 9.9'

# f-strings:
>>> f"x is {x:f} and x^2 is approx {x**2:.1f}"
'x is 3.141593 and x^2 is approx 9.9'
>>> f"{x=:f} and {x**2=:.1f}"  # alternative simplification
'x=3.141593 and x**2=9.9'
```

What formatting should I use?

- use f-strings if you can
- The `.format` method more elegant and versatile than %
- `%` operator style okay, links to Matlab, C, ...
- Choice partly a matter of taste, history and existing code
  - do your collaborators know the method you use?
  - Should be aware (in a passive sense) of different possible styles (so we can read code from others)

Random other things

Changes from Python 2 to Python 3: print

One (maybe the most obvious) change going from Python 2 to Python 3 is that the `print` command loses its special status. In Python 2, we could print ”Hello World” using

```
print "Hello World"          # allowed in Python 2
```

Effectively, we call the function `print` with the argument `"Hello World"`. All other functions in Python are called such that the argument is enclosed in parentheses, i.e.

```
print("Hello World")         # required in Python 3
```

This is the new convention *required* in Python 3 (and *allowed* for recent version of Python 2.x.)

```
commit ade29e25ca763e88f5797fc69a7a31744a06e202
Author: Hans Fangohr <fangohr@users.noreply.github.com>
Date:   Mon Feb 17 17:53:59 2025 +0100

    review integration and ODE sections
```